$P4_{16}$ Language Specification

(version 2.0.0-draft)

The P4.org language consortium

2017-05-10

Abstract

P4 is a language for programming the data plane of network devices. This document provides a precise definition of the $P4_{16}$ language, which is the 2016 revision of the P4 language http://p4.org. The primary target audience for this document includes developers who want to write compilers/simulators/IDEs/debuggers for P4 programs. This document may also be useful for P4 programmers who are interested in understanding the language syntax and semantics at a deeper level.

Contents

1. Scope

This specification establishes the form and interpretation of programs, written in the $P4_{16}$ programming language. It specifies:

- The representation of P4 programs
- The syntax and constraints of the P4 language
- The semantic rules for interpreting P4 programs
- The restrictions and limitations of conformant P4 implementations

This specification does not specify:

- The mechanism by which P4 programs are transformed for use on packet processing systems
- The mechanism by which P4 programs are loaded and executed on the packet processing systems
- The mechanism by which input data are delivered to the P4 program
- The mechanism by which output data, produced by P4 program is delivered to the next consumer
- The mechanism by which the control plane can manage stateful objects defined by a P4 program
- The size or complexity of a program and its data that will exceed the capacity of any specific packet processing system or the capacity of a particular target
- All minimal requirements of a packet processing system that is capable of supporting a conforming implementation

2. Terms, definitions and symbols

Throughout this specification, the following terms will be used. Terms explicitly defined in this specification are not to be presumed to refer implicitly to similar terms defined elsewhere. Terms not explicitly defined in this specification should be interpreted according to ISO/IEC 2382:2015 (Information Technology - Vocabulary) or the other generally recognizable sources, such as RFCs.

- **Architecture**: A set of P4-programmable components on the target and their external data plane interfaces.
- **Control Plane**: A class of algorithms and the corresponding input and output data that are concerned with the configuration and the provisioning of the data plane.
- **Data Plane**: A class of algorithms, describing handling of individual packets as they pass through a packet processing system.
- **Deparser**: A part of a P4 program that assembles the outgoing packet headers.
- **Intrinsic Metadata**: A set of data that a P4-programmable component can use to interface with the other components in the system.

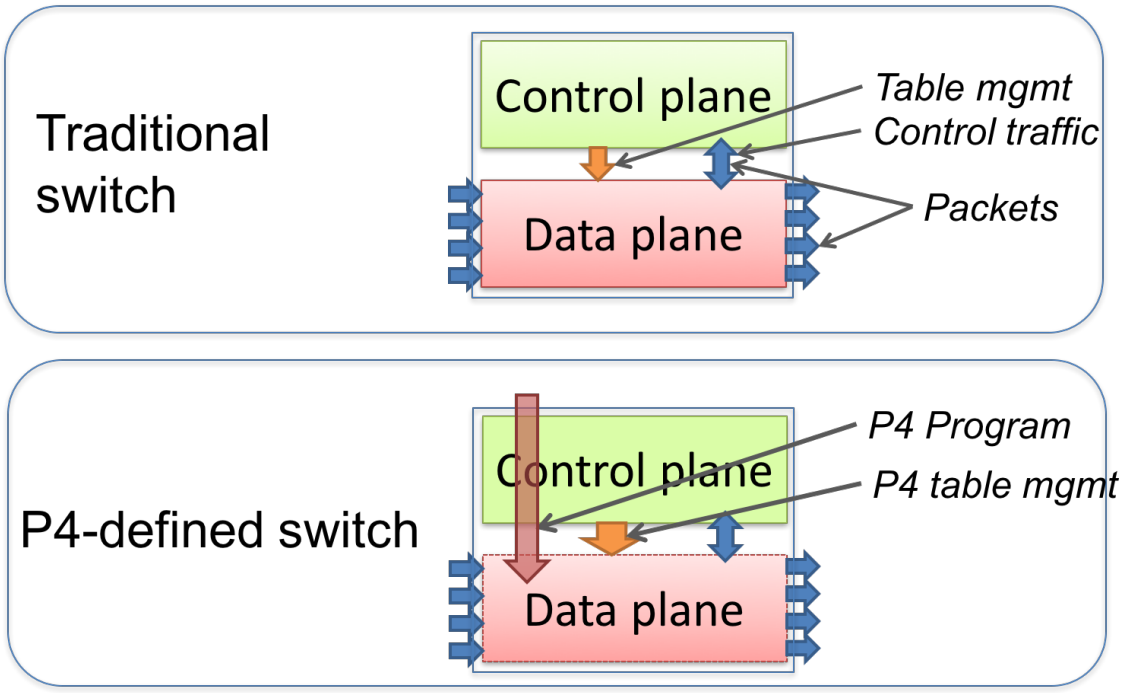

Figure 1. Traditional switches vs. programmable switches.

- **Metadata**: Intermediate data, generated during execution of a P4 program.
- **Packet**: A network packet is a formatted unit of data carried by a packet-switched network.
- **Packet Header**: Packet header refers to supplemental, formally defined data placed at the beginning of a packet. Packet headers are often referred to as packet metadata. A given packet can contain a sequence of packet headers representing different protocols.
- **Packet Payload**: Packet data that follows the Packet Headers.
- **Packet Processing System**: A data processing system oriented towards processing network packets. In general, packet processing systems implement two classes of algorithms, called control plane and data plane.
- **Target**: A Packet Processing System that can execute a P4 program.

3. Overview

P4 is a language for expressing how packets are processed by the data plane of a programmable network forwarding element such as a programmable hardware or software switch, network interface card, router or network function appliance. The name comes from the original paper that introduced the language, "Programming Protocol-independent Packet Processors," https://arxiv.org/pdf/1312.1719.pdf. While initially P4 was designed for programming network switches, its scope has been broadened to cover a large variety of packet processing systems. In the rest of this document we use the generic term *target* for all such network processing system devices.

Many target devices contain both a control plane and a data plane. P4 is designed to specify only the target data plane functionality. P4 programs also partially define the interface by which the control plane and the data-plane communicate, but P4 cannot be used to describe the target's control-plane functionality. In the rest of this document, when we talk about P4 as "programming a target", we mean "programming the target data plane".

As a concrete example in the context of programming network switches, Figure 1 illustrates

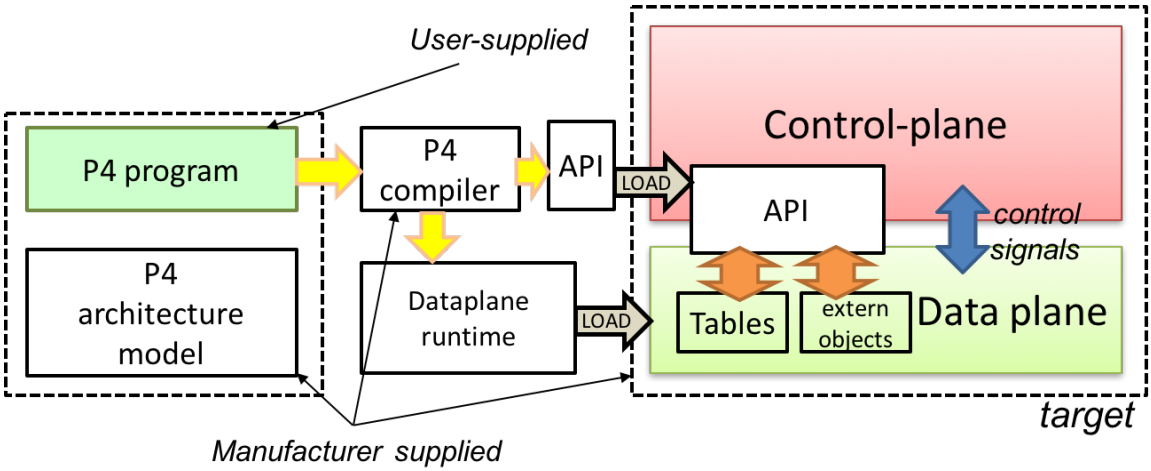

Figure 2. Programming a target with P4.

the difference between a traditional fixed-function switch and a P4-programmable switch. In a traditional switch the manufacturer defines the data-plane functionality. The control-plane controls the data plane by managing entries in tables (e.g. routing tables), configuring specialized objects (e.g. meters) and by processing control-packets (e.g. routing protocol packets) or asynchronous events, such as link state changes or learning notifications.

A P4-programmable switch differs from a traditional switch in two ways:

- The switch data plane is no longer fixed; P4 programs describe the data plane functionality. The data plane is configured at switch initialization time based on the P4 functionality (shown by the long red arrow). The data plane has no built-in knowledge of existing protocols.
- The control plane continues to interact with the data plane using the same channels; however, the set of tables and other objects driving the behavior of the data plane is no longer fixed, since the P4 programmer specifies it after the switch has been manufactured. The P4 compiler generates the API that the control plane uses to communicate with the data plane from the P4 program.

Hence, P4 itself is protocol independent but allows programmers to express a rich set of data plane behaviors and protocols.

The core abstractions provided by the P4 language are:

- **Header definitions** describe the format (the set of fields and their sizes) of each header within a packet.
- **Parsers** describe the permitted header sequences within received packets, how to identify those header sequences, and the headers and fields to extract from packets.
- **Tables** associate user-defined keys with actions. P4 tables generalize traditional switch tables; they can be used to implement routing tables, flow lookup tables, access-control lists, and other user-defined table types, including complex multivariable decisions.
- **Actions** are code fragments that describe how packet header fields and metadata are manipulated. Actions can also include data, which can be supplied by the control-plane at run time.
- **Match-action units** perform the following sequence of operations:
  - Construct lookup keys from packet fields or computed metadata,
  - Perform table lookup using the constructed key, choosing an action (including the associated data) to execute

  - Finally, execute the selected action

- **Control flow** expresses an imperative program describing the data-dependent packet processing within a target pipeline, including the data-dependent sequence of match-action unit invocations. Deparsing (packet reassembly) can be performed using a control flow.
- **Extern objects** are library constructs that can be manipulated by P4 programs through well-defined APIs, but whose internal behavior is hardwired (e.g., checksum units) and hence not programmable using P4.
- **User-defined metadata**: user-defined data structures associated with each packet.
- **Intrinsic metadata**: metadata provided by the architecture associated with each packet (e.g., the input port where a packet has been received).

Figure 2 shows a typical tool workflow when programming a target using P4.

Target manufacturers provide the hardware or software implementation framework, an architecture definition, and a P4 compiler for that target. P4 programmers write programs in P4 for a specific architecture. The *architecture* defines a set of P4-programmable components on the target as well as their external data plane interfaces.

Compiling a set of P4 programs produces two artifacts:

- a data plane configuration that implements the forwarding logic described in the input P4 program and
- an API for managing the behavior of the data plane objects from the control plane

P4 is a domain-specific language. It is designed to be implementable on a large variety of target platforms such as programmable network cards, FPGA switches, software switches and programmable ASICs. As such the language is restricted to constructs that are efficiently implementable on all these platforms.

P4 is not a Turing-complete language. In fact, assuming a fixed cost for a table lookup operation, all P4 program blocks (i.e., parser or control) should provably execute a constant O(1) number of operations for each byte of an input packet received (i.e., each header byte analyzed). In other words, the computational complexity of a P4 program depends linearly only on the header sizes, and never depends on the size of the state accumulated while processing data (e.g., the number of flows, or the total number of packets processed). These guarantees are necessary (but not sufficient) for enabling fast packet processing across a variety of targets.

P4 conformance of a target is defined as follows: if a specific target T supports only a subset of the P4 programming language (let's call it $P4^T$), the programs written in $P4^T$ executed on the target should provide the exact same behavior as P4 programs executed on the P4 abstract machine in this document.

Note that P4 conformant targets can provide arbitrary P4 language extensions and `extern` elements.

3.1. Benefits of P4

Compared to the state-of-the-art method of programming network processing systems (e.g., writing microcode on top of custom hardware), P4 provides significant advantages:

- **Flexibility**: P4 makes many packet-forwarding policies expressible as programs; contrast this to traditional switches, which expose fixed-function forwarding engines to their users
- **Expressiveness**: P4 programs may express sophisticated hardware-independent packet processing algorithms using solely general-purpose operations and table look-ups. Such programs will be portable between hardware targets that implement similar architectures (assuming enough resources are available).
- **Resource mapping and management**: P4 programs express resource usage in symbolic terms (e.g., IPv4 source address); compilers map such user-defined fields to available hardware resources and manage resource allocation and scheduling.

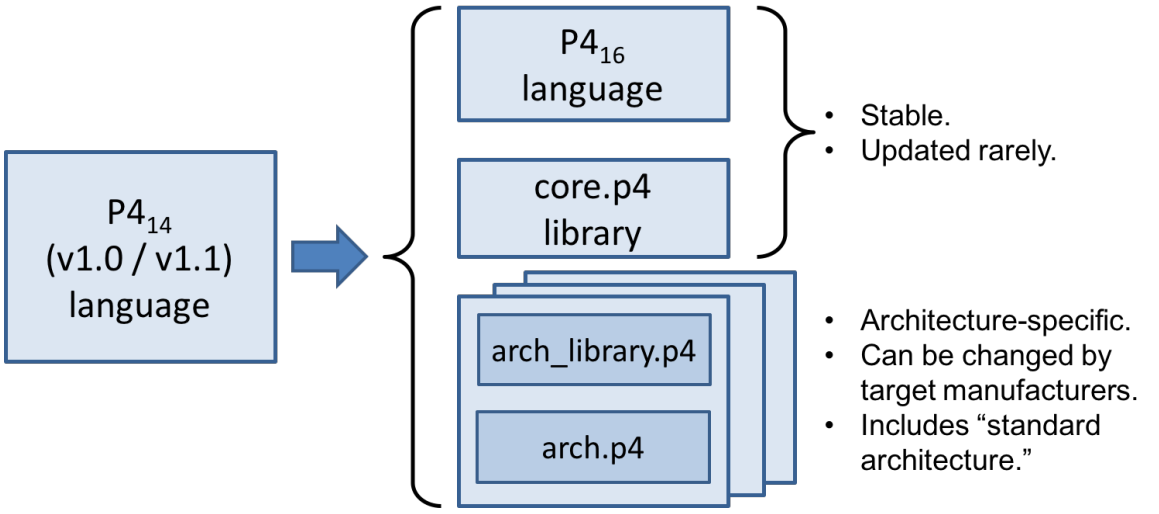

Figure 3. Evolution of the language between versions $P4_{14}$ (versions 1.0 and 1.1) and $P4_{16}$.

- **Software engineering**: P4 programs provide important benefits such as type checking, information hiding and software reuse.
- **Component libraries**: Manufacturer-supplied component libraries may be used to wrap hardware-specific functions into portable high-level P4 constructs.
- **Decoupling hardware and software evolution**: Target manufacturers may use abstract architectures to further decouple the evolution of low-level architectural details from high-level processing.
- **Debugging**: Manufacturers can provide their customers software models of an architecture to aid in the development and debugging of P4 programs.

3.2. P4 language evolution: comparison to previous versions (P4 v1.0/v1.1)

The P4 language went through some significant, backwards-incompatible changes to the language syntax and semantics. The language evolution between the previous version ($P4_{14}$) and the current one ($P4_{16}$) is illustrated in Figure 3. In particular, a significant number of language constructs have been eliminated from the language and will be migrated into libraries. Examples include: counters, checksum units, meters, etc.

The original complex language (74 keywords) has been thus transformed into a relatively small core language (35 keywords, shown in Section A) accompanied by a core library of fundamental constructs which are necessary for writing almost all P4 programs. This core language is the subject of this specification.

The v1.1 version of P4 introduced a language construct called `extern` which can be used to describe library elements. Many constructs that are defined in the v1.1 language specification will thus be transformed into such library elements (including equivalents to v1.1 constructs that have been eliminated from the language, such as counters and meters). Some of these `extern` objects are expected to be standardized, and they will be in the scope of a separate document, describing a standard library of P4 elements. In this document we provide several examples of `extern` constructs.

The $P4_{16}$ language also introduces and repurposes some v1.1 language constructs for describing the programmable parts of an architecture. These language constructs are: `parser`, `state`, `control`, and `package`.

One important goal of the $P4_{16}$ language revision is to provide a *stable* programming language definition, that promotes backwards-compatibility. In other words, we strive for programs written in $P4_{16}$ to remain syntactically correct and to behave identically when treated as programs for later

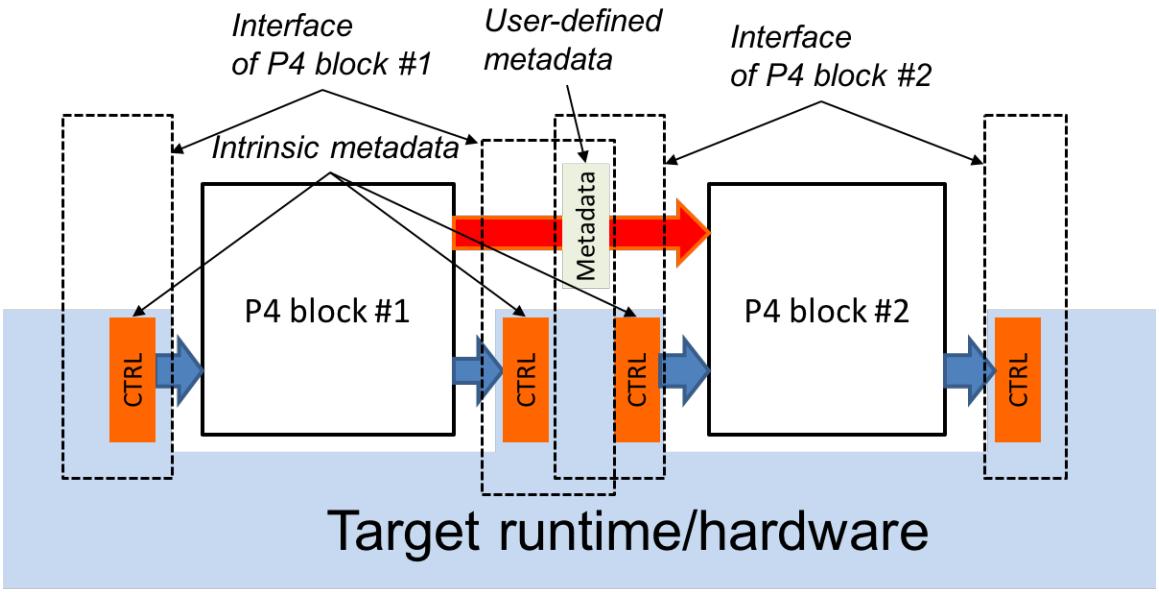

Figure 4. P4 program interfaces.

versions of the P4 language.

4. Architecture Model

4.1. The architecture

The *P4 architecture* identifies the P4-programmable blocks (e.g., parser, ingress pipeline, egress pipeline, deparser, etc.) and their data plane interfaces.

The P4 architecture can be thought of as a contract between the target and P4 code, executing on it. Each target manufacturer must therefore provide both the P4 compiler for it as well as an accompanying architecture definition for their target. (We expect that P4 compilers for all architectures can share a common front-end). This architecture definition does not have to expose the entire programmable surface of the data plane — a manufacturer may even choose to provide multiple definitions for the same hardware device, each with different capabilities (e.g., with or without multicast support).

Figure 4 illustrates the data plane interfaces of P4 programs. It shows a target that has two programmable blocks (#1 and #2). Each block is programmed through a dedicated P4 program fragment. The target interfaces with the P4 program through a set of control registers or signals. Input controls provide information to P4 programs (e.g., the input port that a packet was received from), while output controls can be written to by P4 programs to influence the target behavior (e.g., the output port where a packet has to be directed). Control registers/signals are represented in P4 by *intrinsic metadata*.

Moreover, P4 programs can store and manipulate data pertaining to each packet, represented by *user-defined metadata*.

The behavior of each P4 program is completely described as a set of transformations mapping vectors of bits to vectors of bits. However, *only the architecture model attaches a meaning to the bits in a control register*. For example, in order to cause a network packet to be forwarded on a specific output port, a P4 program may need to write the output port index into a dedicated control register. Similarly, in order to cause a packet to be dropped, a P4 program may need to set a "drop"

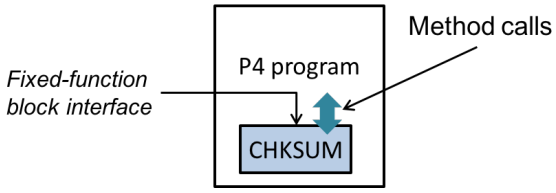

Figure 5. P4 program invoking the services of a fixed-function object.

bit into another dedicated control register.

P4 programs can invoke services of fixed-function blocks. Figure 5 shows a P4 program invoking the services of a target built-in checksum computation unit. The implementation of the checksum unit is not specified in P4, but its interface is. Interfaces (represented by P4 `extern` objects) describe the set of operations that are offered by a fixed-function object as well as their arguments, similar to methods in object-oriented programming languages.

In general, P4 programs are not expected to be portable across different architectures. For example, executing a P4 program that controls packet broadcast by writing into a custom control register will not work on a target that provides no such control register. However, P4 programs written for a given architecture should be portable across all targets that faithfully implement the corresponding model (assuming that enough resources are available to implement the program).

4.2. The P4 standard architecture

We expect that the P4 community will evolve a standard architecture model (or a small set of models, pertaining to specific verticals, e.g., switches and network cards). Wide adoption of such standard architectures should promote wide portability of P4 programs. The standard architecture is the scope of a separate document.

4.3. P4 program data plane interfaces

(In the following examples P4 keywords are highlighted in fixed-size fonts, e.g., `parser`.) To describe a functional block that can be programmed in P4 the target manufacturer provides a target type declaration. Target declarations are discussed in Section 15, but we give a brief introduction here to make things concrete. For example, the target manufacturer could write a declaration as follows:

```
control matchActionPipe<H>(in bit<4> inputPort,
                           inout H parsedHeaders,
                           out bit<4> outputPort);
```

This declaration describes a programmable block named `matchActionPipe` that performs match-action processing (shown by the `control` P4 reserved keyword). The interface between the `matchActionPipe` block and the surrounding hardware can be read from this declaration:

- `bit<4>` is a type indicating a 4-bit value,
- The `in`, `inout`, and `out` keywords indicate the direction of parameters
- The first input is a 4-bit value named `inputPort`, received as an input (`in`) (an instance of intrinsic metadata).
- The second input is an object of type `H`, named `parsedHeaders`. It is both an input and an output, shown by `inout`.
- The type `H` is given by a type variable `<H>` in the declaration (the syntax is similar to Java). The type `H` indicates a type that will be defined later by the programmer.
- The first output is the same `parsedHeaders` as an output, also stored in general-purpose registers. The user mutates this value in-place in the pipeline implementation, and the value of the `parsedHeaders` at the completion of pipe is the result of the computation.

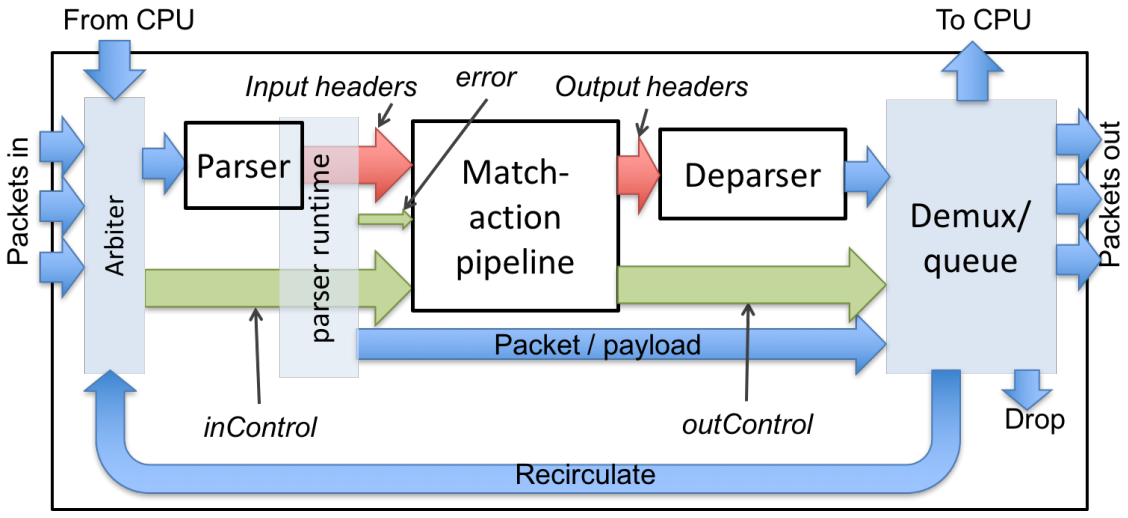

Figure 6. The Very Simple Switch (VSS) architecture.

- The second output is a four-bit value named `outputPort`. This value is written into a control register.

4.4. External units with predefined functionality

P4 programs can also interact with fixed-function objects by invoking their services. Such fixed-function objects are described using the `extern` object construct, which only describes the interfaces that such an object exposes to the data-plane; a complete description of `extern` is given in Section 7.2.8, but we provide an overview here.

`extern` objects are similar to pure abstract classes from object-oriented programming (or interfaces in Java and C#), by describing a set of methods that are implemented by an object, but not the implementation of these methods. For example, the following construct could be used to describe the operations offered by an incremental checksum unit:

```
extern Checksum16 {
    Checksum16();              // constructor
    void clear();              // prepare unit for computation
    void update<T>(in T data); // add data to checksum
    void remove<T>(in T data); // remove data from existing checksum
    bit<16> get(); // get the checksum for the data added since last clear
}
```

5. Example: A very simple switch

To make the discussion concrete we begin by presenting a complete example: we start by describing the architecture of a very simple switch. We then provide the P4 description of the architecture. We end by writing a complete P4 program for controlling the switch. While the example is relatively involved, it makes use all important features of the P4 programming language.

We call our architecture the "Very Simple Switch" (VSS). Figure 6 is a diagram of this architecture. There is nothing special about VSS, it is just a pedagogical example which allows us to illustrate how programmable switches can be described and programmed in P4. We expect that each target manufacturer will publish one or multiple architecture model(s) reflecting the capa-

bilities of their own hardware; in addition we expect that the community will evolve a standard switch architecture, which will be more full-featured than VSS. VSS has some fixed-function blocks (shown in cyan in our example), whose behavior is described in Section 5.2. The white blocks are programmable using P4.

VSS receives packets through one of 8 input Ethernet ports, through a recirculation channel, or from a port connected directly to the CPU. VSS has one single parser, feeding into a single match-action pipeline, which feeds into a single deparser. After exiting the deparser, packets are emitted through one of 8 output Ethernet ports or one of 3 "special" ports:

- Packets sent to the "CPU port" are sent to the control plane
- Packets sent to the "Drop port" are discarded
- Packets sent to the "Recirculate port" are re-injected in the switch through a special input port

Packets may be received from one of three sources:

- One of 8 input ports
- Through recirculation
- From the control plane CPU

The white blocks in the figure are programmable, and the user must provide a corresponding P4 program to control the behavior of each such block. The cyan blocks are fixed-function. The green arrows are data plane interfaces used to convey information between the fixed-function blocks and the programmable blocks — exposed in the P4 program as intrinsic metadata. The red arrows show the flow of user-defined data. VSS has three programmable blocks: a parser, a match-action pipeline, and a deparser.

5.1. Very Simple Switch Architecture

The following P4 program provides a declaration of VSS in P4, as it would be provided by the VSS manufacturer. The declaration contains several type declarations, constants, and finally declarations for the three programmable blocks; the code uses syntax highlighting. The programmable blocks are described by their types; the implementation of these blocks has to be provided by the switch programmer.

```
// File "very_simple_model.p4"
// Simple switch P4 declaration
// core library needed for packet_in definition
#include <core.p4>
/* Various constants and structure declarations */
/* ports are represented using 4-bit values */
typedef bit<4> PortId;
/* only 8 ports are "real" */
const PortId REAL_PORT_COUNT = 4w8;  // 4w8 is the number 8 in 4 bits
/* metadata accompanying an input packet */
struct InControl {
    PortId inputPort;
}
/* special input port values */
const PortId RECIRCULATE_IN_PORT = 0xD;
const PortId CPU_IN_PORT = 0xE;
/* metadata that must be computed for outgoing packets */
struct OutControl {
    PortId outputPort;
}
```

```
/* special output port values for outgoing packet */
const PortId DROP_PORT = 0xF;
const PortId CPU_OUT_PORT = 0xE;
const PortId RECIRCULATE_OUT_PORT = 0xD;
/* Prototypes for all programmable blocks */
/**
 * Programmable parser.
 * @param <H> type of headers; defined by user
 * @param b input packet
 * @param parsedHeaders headers constructed by parser
 */
parser Parser<H>(packet_in b,
                 out H parsedHeaders);
/**
 * Match-action pipeline
 * @param <H> type of input and output headers
 * @param headers headers received from the parser and sent to the deparser
 * @param parseError error that may have surfaced during parsing
 * @param inCtrl information from architecture, accompanying input packet
 * @param outCtrl information for architecture, accompanying output packet
 */
control Pipe<H>(inout H headers,
                in error parseError,// parser error
                in InControl inCtrl,// input port
                out OutControl outCtrl); // output port
/**
 * Switch deparser.
 * @param <H> type of headers; defined by user
 * @param b output packet
 * @param outputHeaders headers for output packet
 */
control Deparser<H>(inout H outputHeaders,
                    packet_out b);
/**
 * Top-level package declaration - must be instantiated by user.
 * The arguments to the package indicate blocks that
 * must be instantiated by the user.
 * @param <H> user-defined type of the headers processed.
 */
package VSS<H>(Parser<H> p,
               Pipe<H> map,
               Deparser<H> d);
// Architecture-specific objects that can be instantiated
// Checksum unit
extern Checksum16 {
    Checksum16();              // constructor
    void clear();              // prepare unit for computation
    void update<T>(in T data); // add data to checksum
    void remove<T>(in T data); // remove data from existing checksum
    bit<16> get(); // get the checksum for the data added since last clear
}
```

Let us describe some of these elements:

- The included file `core.p4` is described in more detail in Appendix B. We use it here for some standard data-types and error codes.
- `bit<4>` is the type of bit-strings with 4 bits.
- The syntax `4w0xF` indicates the value 15 represented using 4 bits. An alternative notation is `4w15`. In some circumstances the width modifier can be omitted, writing just `15`.
- `error` is a built-in P4 type for holding error codes
- Next follows the declaration of a parser:

```
parser Parser<H>(packet_in b, out H parsedHeaders);
```

This declaration describes a parser — but not yet its implementation. The parser implementation will have to be provided by the user. The parser reads its input from a `packet_in`, which is a pre-defined P4 abstraction that represents an incoming network packet, declared in the `core.p4` library. The parser writes its output (the `out` keyword) into the `parsedHeaders` argument. The type of this argument is `H`, yet unknown - it will also be provided by the user.

- The declaration

```
control Pipe<H>(inout H headers,
                in error parseError,
                in InControl inCtrl,
                out OutControl outCtrl);
```

  describes the interface of a Match-Action pipeline named `Pipe`.

The pipeline receives 3 inputs: the headers `headers`, a parser error `parseError`, and the `inCtrl` control data. Figure 6 indicates the different sources of these pieces of information. The pipeline writes its outputs into `outCtrl`, and it must update in place the headers to be consumed by the deparser.

- The top-level package is called `VSS`; in order to program a VSS, the user will have to instantiate a package of this type (shown in the next section). The top-level package declaration also depends on a type variable H:

```
package VSS<H>
```

A type variable indicates a type yet unknown that must be provided by the user at a later time. In this case `H` is the type of the set of headers that the user program will be processing; the parser will produce the parsed representation of these headers, and the match-action pipeline will update the input headers in place to produce the output headers.

- The `package VSS` declaration has 3 complex parameters, of types `Parser`, `Pipe` and `Deparser` respectively; which are exactly the declarations we have just described. In order to program the target one has to supply values for these parameters.
- In this program the `inCtrl` and `outCtrl` structures represent control registers. The content of the headers structure is stored in general-purpose registers.
- The `extern Checksum16` declaration describes an external block whose services can be invoked to compute checksums.

5.2. Very Simple Switch Architecture Description

In order to fully understand VSS's behavior and write meaningful P4 programs for it, and for implementing a control plane, we also need a full behavioral description of the fixed-function blocks. This section can be seen as a simple example illustrating all the details that have to be handled when writing an architecture description. The P4 language is not intended to cover the description of all such functional blocks — the language can only describe the interfaces between programmable blocks and the target — in the previous program this interface is given by the `Parser`, `Pipe`, and `Deparser` declarations. In practice we expect that the complete target description will be provided as

an executable program and/or diagrams and text; in this document we provide a verbal description in English.

5.2.1. Arbiter block

The input arbiter block performs the following functions:

- It receives packets from one of the physical input Ethernet ports, from the control plane or from the input recirculation port.
- For packets received from Ethernet ports, the block computes the Ethernet trailer checksum and verifies it. If the checksum does not match, the packet is discarded. If the checksum does match, it is removed from the packet payload.
- Receiving a packet involves running an arbitration algorithm if multiple packets are available.
- If the arbiter block is busy processing a previous packet and no queue space is available, input ports may drop arriving packets, without indicating the fact that the packets were dropped in any way.
- After receiving a packet, the arbiter block sets the `inCtrl.inputPort` value that is an input to the parser with the identity of the input port where the packet originated. Physical Ethernet ports are numbered 0 to 7, while the input recirculation port has a number 13 and the CPU port has the number 14.

5.2.2. Parser runtime block

The parser runtime block works in concert with the parser. It provides an error code to the match-action pipeline, based on the parser actions, and it provides information about the packet payload (e.g., the size of the remaining payload data) to the demux block. As soon as a packet's processing is completed by the parser, the match-action pipeline is invoked with the associated metadata as inputs (packet headers and user-defined metadata).

5.2.3. Demux block

The core functionality of the demux block is to receive the headers for the outgoing packet from the deparser and the packet payload from the parser, to assemble them into a new packet and to send the result to the correct output port. The output port is specified by the value of `outCtrl.ouputPort`, which is set by the match-action pipeline.

- Sending the packet to the drop port causes the packet to disappear forever.
- Sending the packet to an output Ethernet port numbered between 0 and 7 causes it to be emitted on the corresponding physical switch interface. The packet may be placed in a queue if the output interface is busy emitting a previous packet. When the packet is emitted, the physical interface computes a correct Ethernet checksum trailer and appends it to the packet.
- Sending a packet to the output CPU port causes the packet to be transferred to the control plane. In this case, **the packet that is sent to the CPU is the original input packet**, and not the packet received from the deparser. The latter packet is discarded.
- Sending the packet to the output recirculation port causes it to appear at the input recirculation port. Recirculation is useful when packet processing cannot be done in a single pass.
- If the `outputPort` has an illegal value (e.g., 9), the packet is sent to the drop port.
- If the demux unit is busy processing a previous packet and there is no capacity to queue the packet coming from the deparser, **the demux unit may drop the packet by its own choice**, irrespective of the output port indicated.

Please note that some of the behaviors of the demux block may be unexpected — we have highlighted them in bold. We are not specifying here several important behaviors related to queue size, arbitration and timing, which also influence the packet processing.

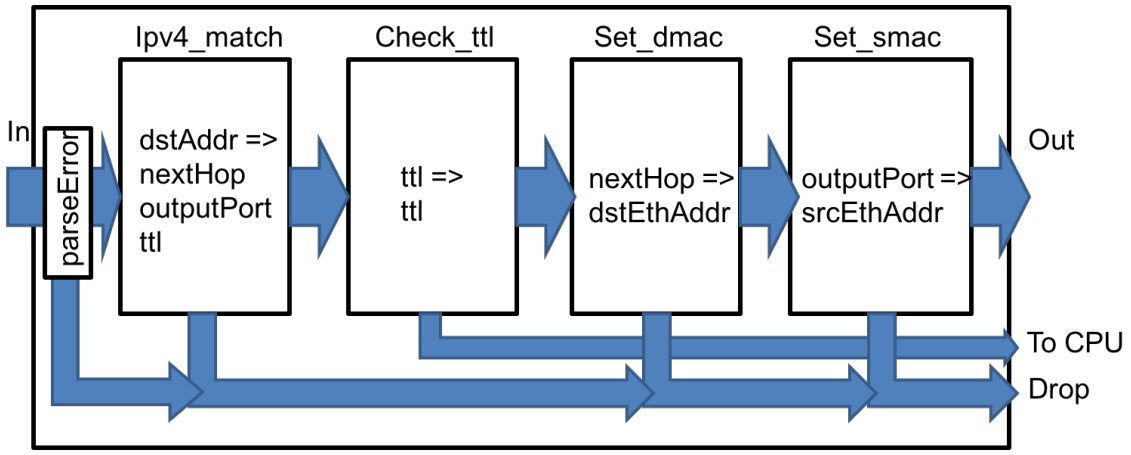

Figure 7. Diagram of the match-action pipeline expressed by the VSS P4 program.

The arrow shown from the parser runtime to the demux block represents an additional information flow from the parser to the demux: the packet being processed as well as the offset within the packet where parsing ended (i.e., the start of the packet payload).

5.2.4. Available extern blocks

The VSS architecture provides an incremental checksum extern block, called `Checksum16`. The checksum unit has a constructor and four methods:

- `clear()` — prepares the unit for a new computation
- `update<T>(in T data)` — add some data to be checksummed. The data must be either a bit-string, a header-typed value, or a `struct` containing such values. The fields in the header/struct are concatenated in the order they appear in the type declaration.
- `get()` — returns the 16-bit one's complement checksum. When this function is invoked the checksum must have received an integral number of bytes of data.
- `remove<T>(in T data)` — under the assumption that `data` was used for computing the current checksum, `data` is removed from the checksum ("undo").

5.3. A complete Very Simple Switch program

Here we provide a complete P4 program performing L2/L3 forwarding for IPv4 packets for the VSS. This program does not take advantage of some features of the switch: e.g., recirculation. The details of many constructs will be explained throughout the document.

Parsing attempts to recognize an Ethernet header and an IPv4 header; if these headers are missing parsing terminates with an error. Otherwise it extracts the information from these headers into a structure with type `Parsed_packet`. The match-action pipeline is shown in Figure 7; it comprises 4 match-action units (represented by the P4 `table` keyword):

- If any parser error has occurred, the packet is dropped (sending it to `DROP_PORT`)
- The first table uses the IPv4 destination address to discover the `outputPort` and the IPv4 address of the next hop. If this look-up fails, the packet is dropped. The table also decrements the IPv4 ttl value.
- The second table checks the ttl value: if the ttl becomes 0, the packet is sent through the CPU port towards the control plane.
- The third table uses the IPv4 address of the next hop (computed by the first table) to figure out the Ethernet address of the next hop.

- Finally, the last table uses the `outputPort` to identify the source Ethernet address of the current switch, which is set in the outgoing packet.

The deparser puts together a packet by reassembling the Ethernet and IPv4 headers as computed in the pipeline. This example uses preprocessor `#include` directives (see Section 6.2).

```
#include <core.p4>

// include the very simple switch declaration from the previous section
#include "very_simple_model.p4"

// This program processes packets composed of an Ethernet and
// an IPv4 header, performing forwarding based on the
// destination IP address

typedef bit<48>  EthernetAddress;
typedef bit<32>  IPv4Address;

// standard Ethernet header
header Ethernet_h {
    EthernetAddress dstAddr;
    EthernetAddress srcAddr;
    bit<16>         etherType;
}

// IPv4 header without options
header IPv4_h {
    bit<4>       version;
    bit<4>       ihl;
    bit<8>       diffserv;
    bit<16>      totalLen;
    bit<16>      identification;
    bit<3>       flags;
    bit<13>      fragOffset;
    bit<8>       ttl;
    bit<8>       protocol;
    bit<16>      hdrChecksum;
    IPv4Address  srcAddr;
    IPv4Address  dstAddr;
}

// Parser section

// Declare user-defined errors that may be signaled during parsing
error {
    IPv4OptionsNotSupported,
    IPv4IncorrectVersion,
    IPv4ChecksumError
}

// List of all recognized headers
struct Parsed_packet {
    Ethernet_h ethernet;
```

```
    IPv4_h     ip;
}

parser TopParser(packet_in b, out Parsed_packet p) {
    Checksum16() ck;  // instantiate checksum unit

    state start {
        b.extract(p.ethernet);
        transition select(p.ethernet.etherType) {
            0x0800: parse_ipv4;
            // no default rule: all other packets rejected
        }
    }

    state parse_ipv4 {
        b.extract(p.ip);
        verify(p.ip.version == 4w4, error.IPv4IncorrectVersion);
        verify(p.ip.ihl == 4w5, error.IPv4OptionsNotSupported);
        ck.clear();
        ck.update(p.ip);
        // Verify that packet checksum is zero
        verify(ck.get() == 16w0, error.IPv4ChecksumError);
        transition accept;
    }
}

// match-action pipeline section
control TopPipe(inout Parsed_packet headers,
                in error parseError,// parser error
                in InControl inCtrl,// input port
                out OutControl outCtrl) {
     IPv4Address nextHop;  // local variable

     /**
      * Indicates that a packet is dropped by setting the
      * output port to the DROP_PORT
      */
      action Drop_action()
      { outCtrl.outputPort = DROP_PORT; }

      IPv4Address nextHop;

     /**
      * Set the next hop and the output port.
      * Decrements ipv4 ttl field.
      * @param ivp4_dest ipv4 address of next hop
      * @param port output port
      */
      action Set_nhop(IPv4Address ipv4_dest,
                      PortId port) {
          nextHop = ipv4_dest;
          headers.ip.ttl = headers.ip.ttl - 1;
```

```
        outCtrl.outputPort = port;
    }

    /**
     * Computes address of next IPv4 hop and output port
     * based on the IPv4 destination of the current packet.
     * Decrements packet IPv4 TTL.
     * @param nextHop IPv4 address of next hop
     */
    table ipv4_match {
        key = { headers.ip.dstAddr: lpm; }  // longest-prefix match
        actions = {
            Drop_action;
            Set_nhop;
        }

        size = 1024;
        default_action = Drop_action;
    }

    /**
     * Send the packet to the CPU port
     */
     action Send_to_cpu()
     { outCtrl.outputPort = CPU_OUT_PORT; }

    /**
     * Check packet TTL and send to CPU if expired.
     */
    table check_ttl {
        key = { headers.ip.ttl: exact; }
        actions = { Send_to_cpu; NoAction; }
        const default_action = NoAction; // defined in core.p4
    }

    /**
     * Set the destination MAC address of the packet
     * @param dmac destination MAC address.
     */
     action Set_dmac(EthernetAddress dmac)
     { headers.ethernet.dstAddr = dmac; }
    /**
     * Set the destination Ethernet address of the packet
     * based on the next hop IP address.
     * @param nextHop IPv4 address of next hop.
     */
     table dmac {
         key = { nextHop: exact; }
         actions = {
             Drop_action;
             Set_dmac;
         }
```

```
        size = 1024;
        default_action = Drop_action;
    }

    /**
     * Set the source MAC address.
     * @param smac: source MAC address to use
     */
     action Set_smac(EthernetAddress smac)
     { headers.ethernet.srcAddr = smac; }

    /**
     * Set the source mac address based on the output port.
     */
    table smac {
         key = { outCtrl.outputPort: exact; }
         actions = {
              Drop_action;
              Set_smac;
        }
        size = 16;
        default_action = Drop_action;
    }

    apply {
        if (parseError != error.NoError) {
            Drop_action();  // invoke drop directly
            return;
        }

        ipv4_match.apply(); // Match result will go into nextHop
        if (outCtrl.outputPort == DROP_PORT) return;

        check_ttl.apply();
        if (outCtrl.outputPort == CPU_OUT_PORT) return;

        dmac.apply();
        if (outCtrl.outputPort == DROP_PORT) return;

        smac.apply();
    }
}

// deparser section
control TopDeparser(inout Parsed_packet p, packet_out b) {
    Checksum16() ck;
    apply {
        b.emit(p.ethernet);
        if (p.ip.isValid()) {
            ck.clear();              // prepare checksum unit
            p.ip.hdrChecksum = 16w0; // clear checksum
            ck.update(p.ip);         // compute new checksum.
```

```
            p.ip.hdrChecksum = ck.get();
        }
        b.emit(p.ip);
    }
}

// Instantiate the top-level VSS package.
// use TopParser for the p Parser, etc.
VSS(TopParser(),
    TopPipe(),
    TopDeparser()) main;
```

6. P4 Language definition

The P4 language can be viewed as having several distinct components, which we describe separately:

- The core language, comprising of types, variables, scoping, declarations, statements, expressions, etc. We start by describing this part of the language.
- A sub-language for expressing parsers, based on state machines (Section 11).
- A sub-language for expressing match-action computations, based on traditional imperative control-flow (Section 12).
- A sub-language for describing architectures (Section 15).

6.1. Syntax and semantics

6.1.1. Grammar

The complete grammar of $P4_{16}$ is given in Appendix E, using the YACC/bison grammar description language. In this text we use the same grammar; we use the following conventions when we provide excerpts from the grammar:

- grammar rules are written using `fixed-size font`
- `UPPERCASE` symbols denote grammar terminals. Grammar fragments will be shown using a special style, as in the following example:

```
p4program
  : /* empty */
  | p4program declaration
  | p4program ';'
  ;
```

Pseudo-code examples (mostly used for describing the semantics of various P4 constructs) are shown with fixed-size fonts as in the following example:

```
ParserModel.verify(bool condition, error err) {
    if (condition == false) {
        ParserModel.parserError = err;
        ParserModel.currentState = reject;
    }
}
```

6.1.2. Semantics and the P4 abstract machines

The P4 semantics is described in terms of abstract machines executing traditional imperative code. There is an abstract machine for each P4 sub-language (parser, control).The abstract machines are described in this text in pseudo-code and English.

P4 compilers can reorganize the generated code in any way as long as the externally visible behaviors of the P4 programs are preserved as described by this specification. The externally visible behavior of a P4 program is defined entirely by:

- The input/output behavior of all P4 blocks, i.e., the values of the outputs computed by a P4 parser/control block given a set of inputs
- The state maintained by extern blocks

6.2. Preprocessing

To aid composition of programs from multiple source files $P4_{16}$ compilers should support the following subset of the C preprocessor functionality:

- `#define` for defining macros without arguments
- `#undef`
- `#if #else #endif #ifdef #ifndef #elif`
- `#include`

The preprocessor will also remove the following sequence of two ASCII characters: backslash newline (ASCII codes 92, 10).

Additional capabilities of the C preprocessor may be supported, but are not guaranteed (e.g., macros with arguments). Similar to C, `#include` can specify a file name either within double quotes or within <>.

```
#include <system_file>
#include "user_file"
```

The difference between the two forms is the order in which the preprocessor searches for header files when the path is incompletely specified.

In addition, P4 compilers should correctly handle #line directives that may be generated during preprocessing. This functionality allows P4 programs to be built from multiple source files, potentially produced by different programmers at different times:

- the P4 core library, produced by the P4 language designers
- the architecture interfaces, specified by the target manufacturer
- target libraries, describing extern blocks provided by the architecture
- user-defined and other libraries of useful components (e.g. standard protocol header definitions)
- the P4 programs that control programmable functional blocks of a target

6.2.1. P4 core library

Similar to the C standard library, the P4 language specification defines a core library that declares useful P4 constructs. A description of the P4 core library is provided in Appendix B. Most P4 programs will include the core library. Including the core library is done with

```
#include <core.p4>
```

6.3. Lexical constructs

All P4 keywords use only ASCII characters. All P4 identifiers must use only ASCII characters. P4 compilers should handle correctly strings containing 8-bit characters in comments and string literals.

P4 is case-sensitive.

Whitespace characters, including newlines are treated as token separators. Indentation is free-form; however, P4 has C-like block constructs, and all our examples use C-like indentation. Tab characters are treated as spaces.

The lexer recognizes the following kinds of terminals:

- `IDENTIFIER` - start with a letter or underscore, and contain letters, digits and underscores
- `TYPE` - identifier that denotes a type name
- `INTEGER` - integer literals
- `DONTCARE` - a single underscore
- Keywords, e.g., `RETURN`. Each keyword terminal corresponds to a language keyword with the same spelling but using lowercase. For example, the `RETURN` terminal corresponds to the `return` keyword.

6.3.1. Identifiers

P4 identifiers may contain only letters, numbers and the underscore character `_`, and must start with a letter or with underscore. The special identifier consisting of a single underscore `_` is reserved to indicate a "don't care" value in several contexts; its type may vary depending on the context. Some reserved keywords (e.g., `apply`) can be used as identifiers if the context makes it unambiguous.

```
nonTypeName
    : IDENTIFIER
    | APPLY
    | KEY
    | ACTIONS
    | STATE
    ;

name
    : nonTypeName
    | TYPE
    ;
```

6.3.2. Comments

Comments are Java style:

- Single-line comments, spanning to the end of line, introduced by `//`
- Multi-line comments, enclosed between `/*` and `*/`
- Nested multi-line comments are not supported.
- JavaDoc comments, starting with `/**` and ending with `*/`

JavaDoc comments are strongly encouraged for the P4 language elements that are used to synthesize the interface with the control-plane: tables and actions.

Comments are token separators: no comments are allowed within a token, e.g. `bi/**/t` is parsed as two tokens, `bi` and `t`, and not as a single `bit` token.

6.3.3. Literal constants

6.3.3.1. Boolean Literals There are two Boolean literal constants: `true` and `false`.

6.3.3.2. Integer literals Integer literals are positive integers of an arbitrary precision. By default, literals are assumed in base 10. To use a different base for the literal, one of the following prefixes must be employed:

- `0x` or `0X` indicates base 16 (hexadecimal)
- `0o` or `0O` indicates base 8 (octal)
- `0b` or `0B` indicates base 2

The width of a numeric literal in bits can be specified by an unsigned number prefix consisting of a number of bits and a signedness indicator:

- `w` indicates unsigned numbers
- `s` indicates signed numbers

Note that, unlike C, a leading zero by itself does not indicate an octal (base 8) constant. The underscore character is considered a digit within number literals but it is ignored when computing the value of the parsed number. This allows long constant numbers to be more easily read by grouping digits together. The underscore cannot be used in the width specification or as the first character of an integer literal. No comments or whitespaces are allowed within a literal. Here are some examples of numeric literals:

```
32w0xFF        // a 32-bit unsigned number with value 255
32s0xFF        // a 32-bit signed number with value 255
8w0b10101010   // an 8-bit unsigned number with value 0xAA
8w0b_1010_1010 // same value as above
8w170          // same value as above
8s0b1010_1010  // an 8-bit signed number with value -86
16w0377        // 16-bit unsigned number with value 377 (not 255!)
16w0o377       // 16-bit unsigned number with value 255 (base 8)
```

6.3.3.3. String literals String literals (string constants) are specified as an arbitrary sequence of 8-bit characters, enclosed within double quote signs " (ASCII code 34). A P4 string starts with a double quote sign and extends to the first double quote sign which is not immediately preceded by an odd number of backslash characters (ASCII code 92). P4 does not make any validity checks on strings (i.e., it does not check that strings represent legal UTF-8 encodings).

Since P4 does not provide any operations on strings, in general the P4 string literals will be passed unchanged through the P4 compiler to other third-party tools or compiler-backends, including the terminating quotes. These tools can define their own handling of escape sequences (e.g., how to specify Unicode characters, or unprintable ASCII characters).

Here are 3 examples of string literals:

```
"simple string"
"string \" with \" embedded \" quotes"
"string with
embedded line terminator"
```

6.4. Naming conventions

P4 provides a rich assortment of types. There are built-in types to represent constructs such as parsers, pipelines, actions, and tables. Base types include bit-strings, numbers, and errors. Users can construct new types based on these: structures, enumerations, headers, header stacks, header unions, etc.

In this document we adopt the Java-like naming guidelines:

- The P4 built-in types all start with lowercase characters and are shown in bold, e.g., `int<20>`
- In all our examples we write user-defined types with an uppercase character, e.g., `IPv4Address`

- Type variables (e.g., used in templates) are always uppercase, e.g., `parser P<H, IH>(...)`
- All variables are named with lowercase names, e.g., `ipv4header`
- Constants are all uppercase, e.g., `CPU_PORT`
- Errors and enumerations have camel-case names, e.g. `PacketTooShort`

6.5. P4 Program structure

A P4 program is a collection of declarations:

```
p4program
    : /* empty */
    | p4program declaration
    | p4program ';'  /* empty declaration */
    ;

declaration
    : constantDeclaration
    | externDeclaration
    | actionDeclaration
    | parserDeclaration
    | typeDeclaration
    | controlDeclaration
    | instantiation
    | errorDeclaration
    | matchKindDeclaration
    ;
```

Empty declarations are indicated by a single semicolon. Allowing empty declarations, denoted by a semicolon, makes it easy to accommodate the habits of C/C++ and Java programmers (some P4 declarations, e.g., `struct`, do not require terminating semicolons).

6.5.1. Scopes

Various constructs act as namespaces that create local scopes for names:

- Derived type declarations (`struct`, `header`, `enum`) introduce local scopes for the field names
- Block statements introduce local lexically-enclosed scopes
- `parser`, `table`, `action`, and `control` blocks introduce local scopes
- A declaration with type variables introduces a new scope for that variable. For example, in the following `extern` declaration, the scope of the type variable `H` extends to the end of the declaration:

  ```
  extern E<H>(...) { ... } // scope of H ends here.
  ```

  The order of declarations is important; with the exception of parser states, all uses of a symbol must follow the symbol's declaration. (This is a change from the $P4_{14}$ specification, which allows declarations in any order. This requirement is similar to the C language, and it significantly simplifies the implementation of compilers for P4, allowing compilers to use additional information about declared identifiers to resolve ambiguities.)

6.5.2. Stateful elements

Most P4 constructs are stateless: given some inputs they produce a result that solely depends on these inputs. There are only two kinds of constructs that may retain information across packets (we call them "stateful"):

- tables. Tables are read-only for the data plane; their contents can only be modified by the control-plane
- extern objects. Some of these objects may be both read and modified by the data plane. All constructs from the $P4_{14}$ language version that represent state (counters, meters, registers) are represented using extern objects in $P4_{16}$.

In P4 all stateful elements must be explicitly allocated at compilation-time through the process called "instantiation".

In addition, parsers, control blocks and packages may contain stateful element instantiations; thus they are also treated as stateful elements (even if they happen to contain no actual state). All stateful elements (extern objects, parsers, controls, packages, but not tables) are represented in P4 by types. In order to use such an object, it must be first instantiated.

Considering the example in Section 5.3, TopParser, TopPipe, TopDeparser, Checksum16 and Switch are types. These are instantiated in the program to be used: there are two instances of Checksum16, one in TopParser and one in TopDeparser, both called ck. The TopParser, TopDeparser, TopPipe and Switch are instantiated at the end of the program, in the declaration of the main instance object, which is an instance of the Switch type (a package).

6.6. L-values

L-values are expressions that can appear on the left side of an assignment operation or as arguments corresponding to out and inout function parameters. An l-value represents a storage reference. The following expressions are legal l-values:

```
prefixedNonTypeName
    : nonTypeName
    | dotPrefix nonTypeName
    ;

lvalue
    : prefixedNonTypeName
    | lvalue '.' member
    | lvalue '[' expression ']'
    | lvalue '[' expression ':' expression ']'
    ;
```

- Identifiers of a base or derived type.
- Structure/header field member access operations (using the dot notation).
- References to elements within header stacks (see Section 8.14): indexing, and references to last and next.
- The result of a bit-slice operator [m:l].

The following is a legal l-value: headers.stack[4].field. Note that method or function calls cannot return l-values.

6.7. Calling convention: call by copy in/copy out

P4 provides multiple constructs for writing modular programs: extern methods, parsers, controls, actions. All these constructs behave similarly to procedures in traditional programming languages:

- They have named and typed parameters.
- They introduce a new local scope for parameters and local variables.
- They allow arguments to be passed by binding them to their parameters.

Invocations are executed using a copy-in/copy-out semantics.

Each parameter is labeled with a direction:

- `in` parameters are read-only. It is an error if an `in` parameter is used on the left-hand side of an assignment or is passed as a non-`in` argument to a callee. `in` parameters are always initialized by copying the value of the corresponding argument at call execution time.
- `out` parameters are uninitialized (parameters of type `header` are set to "invalid"); they are l-values (they can be assigned to — l-values are described in Section 6.6). The argument corresponding to an `out` parameter in a call must be an l-value; after execution of a call, the value of an `out` parameter is copied out to the corresponding argument after completion of the function.
- `inout` parameters are both `in` and `out`. They must be bound to an l-value argument.
- No direction indicates that value of parameter is either:
  - a compile-time known value
  - an action parameter that can only be set by the control plane
  - an action parameter that can be set directly by another calling action; in this case it behaves like an `in` parameter

Arguments are evaluated from left to right, prior to the called function being invoked. The order of evaluation is important when the expression supplied for an argument can have side-effects. Consider the following example:

```
extern void f(inout bit x, in bit y);
extern bit g(inout bit z);
bit a;
f(a, g(a));
```

The evaluation of `g` may mutate its argument `a`, so the compiler has to ensure that the value passed to `f` for its first parameter is not changed by the evaluation of the second argument. The semantics for evaluating a function call is given by the following algorithm (implementations can be different as long as they provide the same result):

1. Arguments are evaluated from left to right as they appear in the function call expression.
2. For each `out` and `inout` argument the corresponding l-value is saved (so it cannot be changed by the evaluation of the following arguments). This is important if the argument contains array indexing operations.
3. The value of each argument is saved into a temporary.
4. The function is invoked with the temporaries as arguments. We are guaranteed that the temporaries that are passed as arguments are never aliased to each other, so this "generated" function call can be implemented using call-by-reference if supported by the architecture.
5. On function return, the temporaries that correspond to `out` or `inout` arguments are copied in order from left to right into the l-values saved in step 2.

According to this algorithm, the previous function call is equivalent to the following sequence of statements:

```
bit tmp1 = a;      // evaluate a; save result
bit tmp2 = g(a);   // evaluate g(a); save result; modifies a
f(tmp1, tmp2);     // evaluate f; modifies tmp1
a = tmp1;          // copy inout result back into a
```

Step 2 in the above algorithm is important; consider the following example:

```
header H { bit z; }
H[2] s;
f(s[a].z, g(a));
```

The evaluation of this call is equivalent to the following sequence of statements:

```
bit tmp1 = a;           // save the value of a
bit tmp2 = s[tmp1].z;   // evaluate first argument
bit tmp3 = g(a);        // evaluate second argument; modifies a
f(tmp2, tmp3);          // evaluate f; modifies tmp2
s[tmp1].z = tmp2;       // copy inout result back; dest is not s[a].z
```

When used as arguments, `extern` objects can only be passed as directionless parameters (see for example the packet argument in the very simple switch example).

Justification

The main reason for using copy-in/copy-out (instead of the more common call-by-reference convention) is for controlling the side-effects of `extern` functions and methods. `extern` functions and methods are the main mechanism by which a P4 program communicates with its environment. With copy-in/copy-out semantics `extern` functions cannot hold references to P4 program objects; this enables the compiler to limit the side-effects that `extern` functions may have on the P4 program both in space (they can only affect out parameters) and in time (side-effects can only occur at function call time).

`extern` functions can be arbitrarily powerful: they can store information in global storage, spawn separate threads, "collude" with each other to share information — but they cannot access any variable in a P4 program. With copy-in/copy-out semantics the compiler can still reason about the P4 program that invokes `extern` functions.

There are additional benefits of a copy-in copy-out semantics:

- This enables P4 to be compiled for architectures that do not support references (e.g., where all data is allocated to named registers. Such architectures may require array indices that appear in a program to be compile-time known values.)
- It simplifies some compiler analyses, since function parameters can never alias to each other within the function body.

```
parameterList
    : /* empty */
    | nonEmptyParameterList
    ;

nonEmptyParameterList
    : parameter
    | nonEmptyParameterList ',' parameter
    ;

parameter
    : optAnnotations direction typeRef name
    ;

direction
    : IN
    | OUT
    | INOUT
    | /* empty */
    ;
```

Here is a summary of the constraints imposed by the parameter directions:

- When used as arguments, extern objects can only be passed as directionless parameters.

- All constructor parameters are evaluated at compilation-time, and in consequence they must all be directionless (they cannot be `in`, `out` or `inout`); this applies to `package`, `control`, `parser` and `extern` objects. Values for these parameters must be specified at compile-time, and must evaluate to compile-time known values.
- For actions all directionless parameters must be at the end of the parameter list. When an action appears in a `table`'s `actions` list, only the parameters with a direction must be bound.
- Actions can also be explicitly invoked using function call syntax, either from a control block or from another action. In this case, values for all action parameters must be supplied explicitly, including values for the directionless parameters. The directionless parameters in this case behave like `in` parameters.

6.8. Name resolution

6.8.1. Lookup in the top-level namespace

Identifiers or type names can be preceded by a dot `.` prefix, which will cause the identifier to be looked-up in the top-level namespace.

```
dotPrefix
    : '.'
    ;
```

```
const bit<32> x = 2;
control c() {
   int<32> x = 0;
   apply {
       x = x + (int<32>).x;  // x is the int<32> local variable,
                             // .x is the top-level bit<32> variable
   }
}
```

6.8.2. Name resolution order

P4 objects that introduce namespaces are organized in a hierarchical fashion. There is a top-level unnamed namespace containing all top-level declarations.

Identifiers prefixed with a dot are always resolved in the top-level namespace.

References to resolve an identifier are attempted inside-out, starting with the current scope and proceeding to all lexically enclosing scopes. The compiler may provide a warning if multiple resolutions are possible for the same name (name shadowing).

```
const bit<4> x = 1;
control p() {
    const bit<8> x = 8;    // x declaration shadows global x
    const bit<4> y = .x;   // reference to top-level x
    const bit<8> z = x;    // reference to p's local x
    apply {}
}
```

6.8.3. Visibility

Identifiers defined in the top-level namespace are globally visible.

Declarations within a `parser` or `control` are private and cannot be referred to from outside of the enclosing `parser` or `control`.

7. P4 data types

$P4_{16}$ is a statically-typed language. Programs that do not pass the type checker are invalid and must be rejected by the compiler. Some values can be converted to a different type using casts. To make user intents clear, implicit casts are only allowed in a few circumstances and the range of casts available is intentionally restricted.

P4 provides a number of base types as well as type operators that construct derived types.

7.1. Base types

P4 supports the following built-in base types:

- The `void` type, which has no values (can be used only in restricted circumstances)
- The `error` type, used to convey errors in a machine-independent, compiler-managed way
- The `match_kind` type, used for describing the implementation of table lookups
- `bool`, representing Boolean values
- Bit-strings of fixed width, denoted by `bit<>`
- Fixed-width signed integers represented using two's complement `int<>`
- Bit-strings of dynamically-computed width with a fixed maximum width `varbit<>`

```
baseType
    : BOOL
    | ERROR
    | BIT
    | BIT '<' INTEGER '>'
    | INT '<' INTEGER '>'
    | VARBIT '<' INTEGER '>'
    ;
```

7.1.1. The void type

The void type is written as `void`. It contains no values. It can only appear in few restricted places in P4 programs.

7.1.2. The error type

The error type contains opaque values that can be used to signal errors. It is written as `error`. New constants of the error type are defined with the syntax:

```
errorDeclaration
    : ERROR '{' identifierList '}'
    ;
```

All `error` constants thus declared are inserted in the `error` namespace, irrespective of the place where an error is defined. `error` is similar to a C/C# enumeration (`enum`) type. A program can contain multiple `error` declarations, which the compiler will merge together. It is an error to declare the same identifier multiple times. Expressions of type `error` are described in Section 8.2.

For example, the following declaration creates two constants of `error` type (these declarations are from the P4 core library):

```
error { ParseError, PacketTooShort }
```

The underlying representation of errors is target-dependent.

7.1.3. The match kind type

The match_kind type is very similar to the error type, and to a C enum type. All declared identifiers are inserted in the top-level namespace. It is used to declare a set of names that may be used in a table's key property (described in Section 12.2.1). It is an error to declare the same match_kind identifier multiple times.

```
matchKindDeclaration
    : MATCH_KIND '{' identifierList '}'
    ;
```

The core library contains the following match_kind declaration:

```
match_kind {
   exact,
   ternary,
   lpm
}
```

Architectures may support additional match_kinds. The declaration of new match_kinds can only occur within model description files; P4 programmers cannot declare new match kinds.

7.1.4. The Boolean type

The Boolean type contains two values, false and true. The type is written as bool. Booleans are not integers or bit-strings.

7.1.5. Strings

P4 offers no support for string processing. The only strings that can appear in a P4 program are constant string literals, described in Section 6.3.3.3. String literals can only be used in annotations (described in Section 17). For example, the following annotation indicates that a specific name should be used for a table when generating the control-plane API:

```
@name("acl") table t1 { ...}
```

7.1.6. Integers (signed and unsigned)

P4 supports arbitrary-size integer values. The typing rules for the integer types are chosen according to the following principles:

- **Inspired from C**: Typing of integers is modeled after the well-defined parts of C, expanded to cope with arbitrary fixed-width integers. In particular, the type of the result of an expression only depends on the expression operands, and not on how the result of the expression is consumed.
- **No undefined behaviors**: P4 attempts to remedy the undefined C behaviors. Unfortunately, C has many undefined behaviors, including specifying the size of an integer (int), what results are produced on overflow, and the results produced for some input combinations (e.g., shifts with negative amounts, overflows on signed numbers, etc.). In contrast, P4 computations on integer types have no undefined behaviors.
- **Least surprise**: The P4 typing rules are chosen to behave as closely as possible to traditional well-behaved C programs.
- **Forbid rather than surprise**: Rather than provide surprising or undefined results (e.g., in C comparisons between signed and unsigned integers), we have chosen to forbid expressions with ambiguous interpretations. For example, P4 does not allow binary operations that combine signed and unsigned integers.

The priority of arithmetic operations is also chosen identical to C (e.g., multiplication binds stronger than addition).

7.1.6.1. Portability No P4 target can support all possible types and operations. For example, the following type is legal in P4: `bit<23132312>`, but it is highly unlikely to be supported by any practical targets. Hence, each target can impose restrictions on the types it can support. Such restrictions may include:

- The maximum width supported
- Alignment and padding constraints (e.g., arithmetic may only be supported on widths which are an integral number of bytes).
- Constraints on some operands (e.g., some architectures may only support multiplications with small constants, or shifts with small values).

Target-specific documentation should describe such restrictions, and target-specific compilers should provide clear error messages when such restrictions are encountered. An architecture may reject a well-typed P4 program and still be conformant to the P4 spec. However, if an architecture accepts a P4 program as valid, the runtime program behavior should match this specification.

7.1.6.2. Unsigned integers (bit-strings) An unsigned integer (which we also call a "bit-string") has an arbitrary width, expressed in bits. A bit-string of width `W` is declared as: `bit<W>`. `W` must be a compile-time known value (see Section 16.1) evaluating to a positive integer greater or equal to 0.

Bits within a bit-string are numbered from `0` to `W-1`. Bit `0` is the least significant, and bit `W-1` is the most significant.

For example, the type `bit<128>` denotes the type of bit-string values with 128 bits numbered from 0 to 127, where bit 127 is the most significant.

The type `bit` is a shorthand for `bit<1>`.

P4 architectures may impose additional constraints on bit types: for example, they may limit the maximum size, or they may only support some arithmetic operations on certain sizes (e.g., 16-, 32- and 64- bit values).

All operations that can be performed on unsigned integers are described in Section 8.5.

7.1.6.3. Signed Integers Signed integers are represented using 2's complement. An integer with `W` bits is declared as: `int<W>`. `W` must be a compile-time known value evaluating to a positive integer greater than 1.

Bits within an integer are numbered from `0` to `W-1`. Bit `0` is the least significant, and bit `W-1` is the sign bit.

For example, the type `int<64>` describes the type of integers represented using exactly 64 bits with bits numbered from 0 to 63, where bit 63 is the most significant (sign) bit.

P4 architectures may impose additional constraints on signed types: for example, they may limit the maximum size, or they may only support some arithmetic operations on certain sizes (e.g., 16-, 32- and 64- bit values).

All operations that can be performed on signed integers are described in Section 8.6.

7.1.6.4. Dynamically-sized bit-strings Some network protocols use fields whose size is only known at runtime (e.g., IPv4 options). To support restricted manipulations of such values, P4 provides a special bit-string type whose size is set at runtime, called a `varbit`.

`varbit<W>` denotes a bit-string with a width of at most `W` bits, where `W` must be a positive integer that is a compile-time known value. For example, the type `varbit<120>` denotes the type of bit-string values that may have between 0 and 120 bits. Most operations that are applicable to fixed-size bit-strings (unsigned numbers) *cannot* be performed on dynamically sized bit-strings.

P4 architectures may impose additional constraints on varbit types: for example, they may limit the maximum size, or they may require `varbit` values to always contain an integer number of bytes at runtime.

All operations that can be performed on varbits are described in Section 8.8.

7.1.6.5. Infinite-precision integers The infinite-precision data type describes integers with an unlimited precision. This type is written as `int`.

This type is reserved for integer literals and expressions that involve only literals. No P4 runtime value can have an `int` type; at compile time the compiler will convert all int values that have a runtime component to fixed-width types, according to the rules described below.

All operations that can be performed on infinite-precision integers are described in Section 8.7.

7.1.6.6. Integer literal types The types of integer literals (constants) are as follows:

- A simple integer constant has type `int`.
- A positive integer prefixed with an integer width `N` and the character `w` has type `bit<N>`.
- An integer prefixed with an integer width `N` and the character `s` has type `int<N>`.

The table below shows several examples of integer literals and their types. For additional examples of literals see Section 6.3.3.

Literal	Interpretation
10	Type is `int`, value is 10
8w10	Type is `bit<8>`, value is 10
8s10	Type is `int<10>`, value is 10
2s3	Type is `int<2>`, value is -1 (last 2 bits), overflow warning
1w10	Type is `bit<1>`, value is 0 (last bit), overflow warning
1s10	Error: 1-bit signed type is illegal

7.2. Derived types

Additional types can be created in P4 from base types. Some derived types can be created by programmers explicitly using type constructors. P4 provides the following type constructors:

- `enum`
- `header`
- header stacks
- header unions
- `struct`
- `tuple`
- type specialization
- `extern`
- `parser`
- `control`
- `package`

`header`, `enum`, `struct`, `extern`, `parser`, `control`, and `package` can only be used in type declarations, where they introduce a new name for the type. The type can subsequently be referred to using this identifier.

Other types cannot be declared, but are synthesized by the compiler internally to represent the type of certain language constructs. These types are described in Section 7.2.7: set types and function types. For example, the programmer cannot declare a variable with type "set", but she can write an expression whose value evaluates to a `set` type. These types are used in the type-checking process.

```
typeDeclaration
    : derivedTypeDeclaration
    | typedefDeclaration
    | parserTypeDeclaration ';'
    | controlTypeDeclaration ';'
    | packageTypeDeclaration ';'
    ;

derivedTypeDeclaration
    : headerTypeDeclaration
    | structTypeDeclaration
    | enumDeclaration
    ;

typeRef
    : baseType
    | typeName
    | specializedType
    | headerStackType
    | tupleType
    ;

prefixedType
    : TYPE
    | dotPrefix TYPE
    ;

typeName
    : prefixedType
    ;
```

7.2.1. Enumeration types

An enumeration type is a more restricted version of the C enum. It is defined with the following syntax:

```
enumDeclaration
    : optAnnotations ENUM name '{' identifierList '}'
    ;

identifierList
    : name
    | identifierList ',' name
    ;
```

For example, the declaration

```
enum Suits { Clubs, Diamonds, Hearths, Spades }
```

introduces a new enumeration type, which contains four constants. One of these constants is `Suits.Clubs`.

Annotations, represented by the non-terminal `optAnnotations` are described in Section 17. This

declaration introduces a new identifier in the current scope for naming the created type. The underlying representation of such values is not specified, so their "size" in bits is not specified (it is target-specific). Operations on `enum` values are described in Section 8.3.

7.2.2. Header types

The declaration of a `header` type is given by the following syntax:

```
headerTypeDeclaration
    : optAnnotations HEADER name '{' structFieldList '}'
    ;

structFieldList
    : /* empty */
    | structFieldList structField
    ;

structField
    : optAnnotations typeRef name ';'
    ;
```

where each `typeRef` is restricted to a bit-string type (fixed or variable) or an integer type. This declaration introduces a new identifier in the current scope; the type can be referred to using this identifier. A header is similar to a `struct` in C, containing all the specified fields. In addition, a header also contains a hidden Boolean "validity" field. When the "validity" bit is `true` we say that the "header is valid". When a header is created its "validity" bit is automatically set to `false`. The "validity" bit can be manipulated by using the header methods `isValid()`, `setValid()`, and `setInvalid()`, as described in Section 8.13.

An empty header is acceptable:

```
header Empty_h { }
```

Note that an empty header still contains a validity bit.

Headers that do not contain any `varbit` field are "fixed size". Headers containing `varbit` fields have "variable size". The size (in bits) of a fixed-size header is a constant, and it is simply the sum of the sizes of all component fields (without counting the validity bit). There is no padding or alignment of the header fields. *Individual P4 targets may impose some constraints on header types*, e.g., restricting headers to sizes that are an integer number of bytes.

For example, the following declaration describes a typical Ethernet header:

```
header Ethernet_h {
   bit<48> dstAddr;
   bit<48> srcAddr;
   bit<16> etherType;
}
```

The type can be referred to using the introduced identifier; the following is a variable declaration using the newly introduced type:

```
Ethernet_h ethernetHeader;
```

The P4 parser language uses the `extract` method of a packet to "fill in" the fields of a header from a network packet, as described in Section 11.8. The successful execution of an `extract` operation also sets the validity bit of the extracted header to `true`.

Here is an example of an IPv4 header with variable-sized options:

```
header IPv4_h {
   bit<4>       version;
   bit<4>       ihl;
   bit<8>       diffserv;
   bit<16>      totalLen;
   bit<16>      identification;
   bit<3>       flags;
   bit<13>      fragOffset;
   bit<8>       ttl;
   bit<8>       protocol;
   bit<16>      hdrChecksum;
   bit<32>      srcAddr;
   bit<32>      dstAddr;
   varbit<320>  options;
}
```

As discussed in Section 8.10, headers that contain variable-length fields may need to be parsed in multiple steps by being broken into multiple headers.

7.2.3. Header stacks

A header stack represents an array of headers. A header stack type is defined as:

```
headerStackType
    : typeName '[' expression ']'
    ;
```

where `typeName` is the name of a header type. For a header stack `hs[n]`, the term `n` is the maximum defined size, and must be a positive integer that is a compile-time known value. Nested header stacks are not supported. At runtime a stack contains `n` values with type `typeName`, only some of which may be valid. Expressions on header stacks are discussed in Section 8.14.

For example, the following declarations,

```
header Mpls_h {
    bit<20> label;
    bit<3>  tc;
    bit     bos;
    bit<8>  ttl;
}
Mpls_h[10] mpls;
```

introduce a header stack called `mpls` containing 10 entries, each of type `Mpls_h`.

7.2.4. Header Unions

A header union represents an alternative between different headers. Header unions can be used to represent "options" in protocols like TCP and IP. They also provide hints to P4 compilers that only one alternative will be present, allowing it to conserve resources.

A header union is defined as:

```
headerUnionDeclaration
    : optAnnotations HEADER_UNION name
      '{' structFieldList '}'
    ;
```

This declaration introduces a new type with the specified name in the local scope. Each element of the list of fields used to declare a header union must be a header type. However, the empty list of fields is legal.

As an example, the type `Ip_h` below represents the union of an IPv4 and IPv6 headers:

```
header_union IP_h {
  IPv4_h v4;
  IPv6_h v6;
}
```

7.2.5. Struct types

P4 `struct` types are similar to C/C++ `struct` types. They are defined with the following syntax:

```
structTypeDeclaration
    : optAnnotations STRUCT name '{' structFieldList '}'
    ;
```

This declaration introduces a new type with the specified name in the local scope. An empty struct is legal. For example, the structure `Parsed_headers` below contains the headers supported by a simple parser:

```
header Tcp_h { ... }
header Udp_h { ... }
struct Parsed_headers {
    Ethernet_h ethernet;
    Ip_h       ip;
    Tcp_h      tcp;
    Udp_h      udp;
}
```

The table below lists all types that may appear as members of headers, structs, and tuples. Note that `int` means an infinite-precision integer, without a width specified.

	Container type		
Element type	**header**	**header_union**	**struct or tuple**
`bit<W>`	allowed	error	allowed
`int<W>`	allowed	error	allowed
`varbit<W>`	allowed	error	allowed
`int`	error	error	error
`void`	error	error	error
`error`	error	error	allowed
`match_kind`	error	error	error
`bool`	error	error	allowed
`enum`	error	error	allowed
`header`	error	allowed	allowed
header stack	error	error	allowed
`header_union`	error	error	allowed
`struct`	error	error	allowed
`tuple`	error	error	allowed

The two-argument `extract` method on packets only supports a single `varbit` field in a header.

Rationale: `int<W>` is sufficient to allow a signed integer as a member, and it has easily-determined storage requirements, unlike an infinite precision `int`. `match_kind` values are not useful to store in a variable, as they are only used to specify how to match fields in table search keys, which are all declared at compile time. `void` is not useful as part of another data structure. Headers must have precisely defined formats as sequences of bits in order for them to be parsed or deparsed.

7.2.6. Tuple types

A tuple is similar to a `struct`, in that it holds multiple values. Unlike a `struct` type, tuples have no named fields. The type of tuples with n component types `T1`,...,`Tn` is written as

```
tuple<T1,...,Tn>
```

```
tupleType
    : TUPLE '<' typeArgumentList '>'
    ;
```

Operations that manipulate tuple types are described in Sections 8.10 and 8.11. Tuple types can be converted to structure types that have the same number of fields.

```
struct S { bit<32> a; bit<32> b; }
tuple<bit<32>, bit<32>> x;
x = { 32w25, 32w35 };
S y = x;
```

7.2.7. Synthesized data types

For the purposes of type-checking the P4 compiler can synthesize some type representations which cannot be directly expressed by users. These are described in this section: set types and function types.

7.2.7.1. Set types `set<T>` is a type that describes *sets* of values of type `T`. Set types can only appear in restricted contexts in P4 programs. For example, the range expression `8w5 .. 8w8` describes a set containing the 8-bit numbers 5, 6, 7 and 8, so its type is `set<bit<8>>;`. This expression can be used as a label in a `select` expression (see Section 11.6), matching any value in this range. Set types cannot be named or declared by P4 programmers, they are only synthesized by the compiler internally and used for type-checking. Expressions with set types are described in Section 8.11.

7.2.7.2. Function types Currently function types cannot be created explicitly in P4 programs; they are created by the P4 compiler internally to represent the type of a function, procedure, or method, and they are used for type-checking. We also call the type of a function its signature. Libraries can contain extern function declarations.

For example, the following declaration:

```
extern void random(in bit<5> logRange, out bit<32> value);
```

describes an object `random` which has a function type, representing the following information:

- the result type is `void`
- the function has two inputs
- first input has direction `in`, type `bit<5>`, and name `logRange`
- second input has direction `out`, type `bit<32>`, and name `value`

7.2.8. Extern types

P4 programs can invoke the service of fixed-function blocks. A typical example of such a fixed-function block is a checksum unit. The functionality of such blocks is exposed to P4 programs through `extern` declarations.

P4 supports extern object declarations and extern function declarations.

```
externDeclaration
    : optAnnotations EXTERN nonTypeName optTypeParameters '{' methodPrototypes '}'
    | optAnnotations EXTERN functionPrototype ';'
    ;
```

7.2.8.1. Extern functions An extern function declaration describes a function and its signature, but not the function's implementation.

```
functionPrototype
    : typeOrVoid name optTypeParameters '(' parameterList ')'
    ;
```

For an example of an `extern` function declaration, see Section 7.2.7.2.

7.2.8.2. Extern objects An extern object declaration declares an object and *all methods* that can be invoked to perform computations and to alter the state of the object. Extern object declarations can also optionally declare constructor methods; these must have the same name as the enclosing `extern` type, no type parameters, and no return type. Extern declarations can only appear as allowed by the architecture model and may be specific to a target.

```
methodPrototypes
    : /* empty */
    | methodPrototypes methodPrototype
    ;

methodPrototype
    : functionPrototype ';'
    | TYPE '(' parameterList ')' ';' //constructor
    ;

typeOrVoid
    : typeRef
    | VOID
    | nonTypeName     // may be a type variable
    ;

optTypeParameters
    : /* empty */
    | '<' typeParameterList '>'
    ;

typeParameterList
    : nonTypeName
    | typeParameterList ',' nonTypeName
```

```
    ;
```

For example, the P4 core library introduces two interfaces `packet_in` and `packet_out` used for manipulating network packets (see Sections 13.8 and 16.1). Here is an example showing how operations on a network packet can be invoked:

```
extern packet_out {
    void emit<T>(in T hdr);
}
control d(packet_out b, in Hdr h) {
    apply {
        b.emit(h.ipv4);       // write ipv4 header into output packet
    }                         // by calling emit method
}
```

Functions and methods are the only P4 constructs which support overloading: there can exist multiple methods with the same name in the same scope. Even so, two functions (or methods of an `extern` object) can have the same name only if they have a different number of parameters.

7.2.9. Type specialization

A generic type may be specialized by specifying arguments for its type variables. In cases where the compiler can infer type arguments type specialization is not necessary. When a type is specialized all its type variables must be bound.

```
specializedType
    : prefixedType '<' typeArgumentList '>'
    ;
```

For example, the following extern declaration describes a generic block of registers, where the type of the elements stored in each register is an arbitrary `T`.

```
extern Register<T> {
    Register(bit<32> size);
    T read(bit<32> index);
    void write(bit<32> index, T value);
}
```

The type `T` has to be specified when instantiating a set of registers, by specializing the Register type:

```
Register<bit<32>>(128) registerBank;
```

The instantiation of `registerBank` is made using the `Register` type specialized with the `bit<32>` bound to the `T` type argument.

7.2.10. Parser and control blocks types

Parsers and control blocks types are similar to function types: they describe the signature of parsers and control blocks. Such functions have no return values. Parsers and control block types may be generic (i.e., have type parameters).

The types `parser`, `control`, and `package` cannot be used as the types of arguments for methods, parsers, controls, tables, actions. They *can* be used as types for the arguments passed to constructors (see Section 13).

7.2.10.1. Parser type declarations A parser type declaration describes the signature of a parser. A parser should have at least one argument of type `packet_in`, representing the received packet that is processed.

```
parserTypeDeclaration
    : optAnnotations PARSER name optTypeParameters
     '(' parameterList ')'
    ;
```

For example, the following is a type declaration of a parser type named `P` that depends on a type variable `IH`. The parser that receives as input a `packet_in` value `b` and produces two values:

- A value with a user-defined type `IH`
- A value with a predefined type `Counters`

```
struct Counters { ... }
parser P<IH>(packet_in b,
             out IH packetHeaders,
             out Counters counters);
```

7.2.10.2. Control type declarations A control type declaration describes the signature of a control block.

```
controlTypeDeclaration
    : optAnnotations CONTROL name optTypeParameters
      '(' parameterList ')'
    ;
```

Control type declarations are very similar to parser type declarations.

7.2.11. Package types

A package type describes the signature of a package.

```
packageTypeDeclaration
    : optAnnotations PACKAGE name optTypeParameters
      '(' parameterList ')'
    ;
```

All parameters of a package are evaluated at compilation-time, and in consequence they must all be directionless (they cannot be `in`, `out` or `inout`). Otherwise package types are very similar to parser type declarations. Packages can only be instantiated; they have no runtime behaviors associated.

7.2.12. Don't care types

A don't care (underscore, "`_`") can be used in some circumstances as a type. It should be only used in a position where one could write a bound type variable; it is similar to the Java `?` wildcard type. The underscore can be used to reduce code complexity — when it is not important what the type variable binds to (during type unification the don't care type can unify with any other type). An example in given Section 15.1).

7.3. typedef

`typedef` can be used to give an alternative name to a type.

```
typedefDeclaration
    : TYPEDEF typeRef name ';'
    | TYPEDEF derivedTypeDeclaration name ';'
    | annotations TYPEDEF typeRef name ';'
    | annotations TYPEDEF derivedTypeDeclaration name ';'
    ;
typedef bit<32> u32;
typedef struct Point { int<32> x; int<32> y; } Pt;
typedef Empty_h[32] HeaderStack;
```

All operations that can be executed on the original type can be also executed on the newly created type. This behavior is similar to the C `typedef` keyword.

8. Expressions

This section describes all computations that can be performed in P4, grouped by the type of the values than can be processed.

The grammar for general expressions is given by:

```
expression
    : INTEGER
    | TRUE
    | FALSE
    | STRING_LITERAL
    | nonTypeName
    | '.' nonTypeName
    | expression '[' expression ']'
    | expression '[' expression ':' expression ']'
    | '{' expressionList '}'
    | '(' expression ')'
    | '!' expression
    | '~' expression
    | '-' expression
    | '+' expression
    | typeName '.' member
    | ERROR '.' member
    | expression '.' member
    | expression '*' expression
    | expression '/' expression
    | expression '%' expression
    | expression '+' expression
    | expression '-' expression
    | expression SHL expression      // SHL is <<
    | expression '>''>' expression   // check that >> are contiguous
    | expression LE expression       // LE is <=
    | expression GE expression
    | expression '<' expression
    | expression '>' expression
    | expression NE expression       // NE is !=
    | expression EQ expression       // EQ is ==
```

```
    | expression '&' expression
    | expression '^' expression
    | expression '|' expression
    | expression PP expression       // PP is ++
    | expression AND expression      // AND is &&
    | expression OR expression       // OR is ||
    | expression '?' expression ':' expression
    | expression '<' typeArgumentList '>' '(' argumentList ')'
    | expression '(' argumentList ')'
    | typeRef '(' argumentList ')'
    | '(' typeRef ')' expression
    ;

expressionList
    : expression
    | expressionList ',' expression
    ;

member
    : name
    ;

argumentList
    : /* empty */
    | nonEmptyArgList
    ;

nonEmptyArgList
    : argument
    | nonEmptyArgList ',' argument
    ;

argument
    : expression
    ;

typeArg
    : DONTCARE
    | typeRef
    ;

typeArgumentList
    : typeArg
    | typeArgumentList ',' typeArg
    ;
```

See also the complete P4 grammar in Appendix E.

An additional semantic check is required for the right shift to check that there is no space between the two consecutive greater-than signs >>. This rule is required to allow parsing for both the right shift operators and specialized types, such as in `function<bit<32>>`.

This grammar does not indicate the precedence of the various operators. The precedence follows

exactly the C precedence rules. Concatenation (++) has the same precedence as infix addition. Bit-slicing `a[m:l]` has the same precedence as array indexing (`a[i]`).

In addition to these expressions, `select` expressions (described in Section 11.6) may be used only in parsers.

8.1. Expression evaluation order

Given a complex expression, the order in which sub-expressions are evaluated can be important if these sub-expressions can produce side-effects. P4 expressions are evaluated as follows:

- Boolean operators `&&` and `||` are evaluated short-circuit: the second operand is only evaluated if necessary.
- The conditional operator `?:` evaluates its first argument, and based on its values it evaluates the second or the third.
- All other expressions are evaluated left-to-right as they appear in the source program.
- Function and method calls are evaluated as described in Section 6.7.

8.2. Expressions on error values

Symbolic names declared by an `error` declaration belong to the `error` namespace. The `error` type only supports comparisons for equality and difference. The result of a comparison is a Boolean value.

For example, the following operation tests for the occurrence of an error:

```
error errorFromParser;
...
if (errorFromParser != error.NoError) { ... }
```

8.3. Expressions on enum values

Symbolic names declared by an `enum` do not belong to the top-level namespace, but to a newly introduced namespace.

```
enum X { v1, v2, v3 }
X.v1  // reference to v1
v1    // error - v1 is not in the top-level namespace
```

`enum` values can only be compared for equality/difference using `== and !=`. `enum` values cannot be cast to or from any other types.

When `enum` values appear in the control-plane API the compiler back-end has to choose a suitable serialization data type and representation.

8.4. Expressions on Boolean values

The following operations are provided on Boolean values:

- And, designated by `&&`
- Or designated by `||`
- Negation, designated by `!`
- Equality tests (`==` and `!=`)

Operator precedence is similar to C. Operator evaluation is short-circuit.

There are no implicit casts from bit-strings to Booleans or vice-versa. As a consequence, a C program fragment such as:

```
if (x) ...
```

(for x an integer base type) must be written in P4 as:

```
if (x != 0) ...
```

(see also the discussion on infinite-precision types and implicit casts in Section 8.9.2 for how the 0 in this expression is evaluated).

8.4.1. The conditional operator

The `?:` expression behaves as in C, e.g.:

```
(x == 0) ? e0 : e1;
```

The first argument is Boolean, and the second and third arguments must have the same type. The second and third arguments cannot be both infinite precision integers unless the condition itself can be evaluated at compilation time (this restriction exists in order to allow the width of the result of the conditional operation to be inferred; the type of the result cannot be `int`, which is reserved for integer literals). The conditional operator evaluation is short-circuit: only the selected alternative is evaluated.

8.5. Bit-string (unsigned integer) operations

This section discusses all operations that can be performed on values with `bit<W>` types.

Arithmetic operations "wrap-around", similar to C operations on unsigned values (i.e., representing a large value on W bits will only keep the least-significant W bits of the value). There are no arithmetic exceptions; the runtime result of an arithmetic operation is defined for all combinations of input arguments.

All binary operations (except shifts) require both operands to have the same exact type and width; supplying operands with different widths produces an error at compile time. No implicit casts are inserted by the compiler to equalize the widths. There are no binary operations that combine signed and unsigned values (except shifts). The following operations are provided on Bit-string values:

- Test for equality between bit-strings of the same width, designated by `==`. The result is a Boolean value.
- Test for difference between bit-strings of the same width, designated by `!=`. The result is a Boolean value.
- Unsigned comparisons `<,>,<=,>=`. Both operands must have the same width; the result is a Boolean value.

All the following operations produce bit-string results when applied to bit-strings. All these operations require both operands to have the same width.

- Negation, denoted by unary `-`. Result is computed by subtracting the value from 2^W. The result is always unsigned and it has the same width as the input. The semantics is the same as the C negation of unsigned numbers.
- Unary plus, denoted by `+`. Behaves as a no-op.
- Addition, denoted by `+`. Associative. Result is computed by truncating the result of the addition to the width of the output (similar to C).
- Subtraction, denoted by `-`. Result is unsigned, and has the same type as the operands. Result is computed by adding the negation of the second operand (similar to C).
- Multiplication `*`. Result has the same width as the operands. P4 targets may impose additional restrictions (e.g., may only allow multiplications with powers of two).
- Bitwise "and" between two bit-strings of the same width, designated by `&`
- Bitwise "or" between two bit-strings of the same width, designated by `|`
- Bitwise "complement" of a single bit-string, designated by `~`
- Bitwise "xor" of two bit-strings of the same width, designated by `^`

There are also the following operations:

- Concatenation of two bit-strings, resulting in a bit-string whose length is the sum of the lengths, designated by the infix operator `++`. The left bit-string provides the most significant bits.
- Extraction of a set of contiguous bits (bit slice), designated by `[m:l]`, where `m` and `l` must be positive integers that are compile-time known values, and `m >= l`. The result is a bit-string of width `m - l + 1`, including the bits numbered from `l` (which becomes the LSB of the result) to `m` (the MSB of the result) from the source operand. The conditions `0 <= l < W` and `l <= m < W` are checked statically (where `W` is the length of the source bit-string). Note that both endpoints of the extraction are inclusive. The bounds are required to be compile-time known values so that the result width can be computed at compile time.
- Slices are l-values: one can assign to a slice:

  ```
  e[m:l] = x
  ```

  This statement sets bits `m` to `l` of `e` to the bit-pattern represented by `x`, and leaves all other bits of `e` unchanged.
- Logical shift left and right with a runtime known unsigned integer value (left operand is unsigned, right operand must be either an unsigned number of type `bit<S>` or a non-negative constant integer), designated by `<<` and `>>`. The result has the same type as the left operand. Shifts with an amount greater than the width of the input produce a result with all bits zero.

8.6. Operations on fixed-width signed integers

This section discusses all operations that can be performed on `int<W>` types. An `int<W>` type is a signed integer with `W` bits represented using 2's complement.

"Underflow" or "overflow" produced by arithmetic cannot be detected: operations "wrap around", similar to C operations on unsigned values (i.e., representing a large value on W bits will only keep the least-significant W bits of the value)

There are no arithmetic exceptions; the runtime result of an arithmetic operation is defined for all combinations of input arguments (note that C does not specify the result of overflows on signed integers).

All binary operations (except shifts) require both operands to have the same exact type (signedness) and width; supplying operands with different widths or signedness produces an error at compile time. No implicit casts are inserted by the compiler to equalize the types. There are no binary operations that combine signed and unsigned values (except shifts).

Note that bitwise operations are well-defined, since the representation is mandated to be 2's complement.

The `int<W>` datatype supports the following operations; all binary operations require both operands to have the exact same type. The result always has the same width as the left operand.

- Negation, denoted by unary `-`
- Unary plus, denoted by `+`. Behaves as a no-op.
- Addition, denoted by `+`
- Subtraction, denoted by `-`
- Comparison for equality and inequality `==,!=` producing a Boolean result
- Numeric comparisons `<,<=,>,>=` with a Boolean result
- Multiplication `*`. Result has the same width as the operands. P4 targets may impose additional restrictions (e.g., may only allow multiplications with powers of two).
- Arithmetic shift left and right denoted by `<<` and `>>`. Left operand is signed, right operand must be either an unsigned number of type `bit<S>` or a non-negative constant integer. The result has the same type as the left operand. Shifts with an amount greater or equal to the width of the input are allowed.

8.6.1. A note about shifts

Shifts (on signed and unsigned values) deserve a special discussion for the following reasons:

- As in C, right shift behaves differently for signed and unsigned values: right shift for signed values is an arithmetic shift.
- Shifting with a negative amount does not have a clear semantics: while in C the result is undefined, in P4 the type system makes it illegal to shift with a negative amount.
- In C, shifting with an amount larger or equal to the number of bits has an undefined result (unlike the P4 definition).
- Finally, shifting may require doing work which is exponential in the number of bits of the right-hand-side operand.

Consider the following examples:

```
bit<8> x;
bit<16> y;
... y << x ...
... y << 1024 ...
```

Unlike C, P4 gives a precise meaning shifting with an amount larger than the size of the shifted value.

P4 targets may impose additional restrictions on shift operations:

- Targets may reject shifts by non-constant amounts.
- Targets may reject shifts with large non-constant amounts. For example, a target may forbid shifting an 8-bit value by a non-constant value wider than 3 bits.

8.7. Operations on arbitrary-precision constant integers

The type `int` denotes integer values on which computations are performed with arbitrary precision. Only compile-time known values may have type `int` . They support the following operations:

- Negation, denoted by unary `-`
- Unary plus, denoted by `+`. Behaves as a no-op.
- Addition, denoted by `+`
- Subtraction, denoted by `-`
- Comparison for equality and inequality `==,!=` producing a Boolean result
- Numeric comparisons `<,<=,>,>=` with a Boolean result
- Multiplication `*`
- Integer division between positive values, denoted by `/`, rounded towards 0, as in C
- Modulo between positive values, denoted by `%`
- Arithmetic shift left and right denoted by `<<` and `>>`. Right operand must be a positive number. The result is an `int`. `a << b` is $a \times 2^b$. `a >> b` is $\lfloor a/2^b \rfloor$.

All the operands that participate in an operation must have type `int`; binary operations (except shift) cannot combine int values with fixed-width types. For such expressions the compiler will always insert an implicit cast; this cast will always convert the `int` value to the fixed-width type.

All computations on `int` values are carried without information loss. For example, multiplying two 1024-bit values may produce a 2048-bit value (note that concrete representation of `int` values is not specified). `int` values can be cast to `bit<w>` and `int<w>` values. Casting an `int` value to a fixed-width type will preserve the least-significant bits. If the truncation causes significant bits to be lost, the compiler should emit a suitable warning.

Note: bitwise-operations (`|,&,^,~`) are not defined for `int` values. Division and modulo are illegal for negative values.

8.8. Variable bit-string operations

A variable-size bit-string `varbit` has a maximum size static declared width, and also a dynamic width, which must be at most the static width. Prior to initialization a variable-size bit-string has an unknown dynamic width.

Variable-length bit-strings support a limited set of operations:

- Parser extraction into a variable-sized bit-string using the two-argument `extract` method of a `packet_in` (see Section 11.8.3). This operation sets the dynamic width of the field.
- Assignment to another variable-sized bit-string. The target must have the exact same static width. The assignment sets the dynamic width on the target to be the same as the source dynamic width.
- The `emit` method of a `packet_out` interface to insert a dynamically-sized bit-string with known dynamic width into a packet (see Section 14).

8.9. Casts

P4 supports a very limited range of casts, and casts are only allowed between base types. Most binary operations require both operands to have the exact same type. Some type conversions may require multiple chained casts. While more onerous for the user, this approach has several benefits:

- It makes user intent unambiguous.
- It makes the conversion cost explicit. Some casts involve sign-extensions, and thus require significant computational resources.
- It reduces the number of cases that have to be considered in the P4 specification. Some targets may not support all casts. A cast expression is written as in C: `(typeRef)`

8.9.1. Explicit casts

Here are all legal casts:

- `bit<1> <-> bool`: `0` is `false`, `1` is `true`
- `int<W> -> bit<W>`: preserves all bits unchanged; negative values are reinterpreted as positive values
- `bit<W> -> int<W>`: preserves all bits unchanged; values with the MSB set are reinterpreted as negative values
- `bit<W> -> bit<X>`: if `W > X` this causes truncation, if `W < X` this causes extension with zero bits
- `int<W> -> int<X>`: if `W > X` this causes truncation, if `W < X` this causes extension with the sign bit
- `int -> bit<W>`: Represents the integer value using two's complement on a large enough number of bits and keeps the least-significant `W` bits; overflow should lead to a warning, as will conversion of a negative number
- `int -> int<W>`: Represents the integer value using two's complement on a large enough number of bits and keeps the least-significant `W` bits; overflow should lead to a warning
- Given a type declaration introduced by `typedef S T`, values of types `T` and `S` can be cast back and forth if they represent base types.

8.9.2. Implicit casts

Unlike C, P4 allows a very limited number of implicit casts. The reason is that often the implicit casts have a non-trivial semantics, which is invisible for the programmer.

Implicit casts are allowed in P4 only to convert an '`int` value to a fixed-width type.

Most binary operations that take an `int` and a fixed-width operand will insert an implicit cast to convert the `int` operand to the type of the fixed-width operand.

Consider a program with the following values:

```
bit<8>  x;
bit<16> y;
int<8>  z;
```

The following expressions are translated by the compiler as follows:

- `x + 1` becomes `x + (bit<8>)1`
- `z < 0` becomes `z < (int<8>)0`
- `x << 13` becomes `0`; overflow warning
- `x | 0xFFF` becomes `x | (bit<8>)0xFFF`; overflow warning

8.9.3. Illegal arithmetic expressions

Consider a program with the following values:

```
bit<8>  x;
bit<16> y;
int<8>  z;
```

The table below shows several expressions which are illegal because they do not obey the P4 typing rules. For each expression we provide several ways that the expression could be manually rewritten into a legal expression. Note that for some expression there are several legal alternatives, which may produce different results! The compiler cannot guess the user intent, so the user is required to explicitly state it.

Expression	Why it is illegal	Alternatives
`x + y`	Different widths	`(bit<16>)x + y` `x + (bit<8>)y`
`x + z`	Different signs	`(int<8>)x + z` `x + (bit<8>)z`
`(int<8>)y`	Cannot change both sign and width	`(int<8>)(bit<8>)y` `(int<8>)(int<16>)y`

`y + z`	Different widths and signs	`(int<8>)(bit<8>)y + z`
		`y + (bit<16>)(bit<8>)z`
		`(bit<8>)y + (bit<8>)z`
		`(int<16>)y + (int<16>)z`

`x << z`	RHS of shift cannot be signed	`x << (bit<8>)z`
`x < z`	Different signs	`X < (bit<8>)z` `(int<8>)x < z`
`1 << x`	Width of `1` is unknown	`32w1 << x`
`~1`	Bitwise operation on int	`~32w1`
`5 & -3`	Bitwise operation on int	`32w5 & -3`

8.10. List expressions

A list expression is a list of expressions enclosed within curly braces and separated by commas:

```
expression ...
    | '{' expressionList '}'

expressionList
    : expression
    | expressionList ',' expression
    ;
```

The type of a list expression is a tuple type (Section 7.2.7). List expressions can be assigned to `tuple`, `struct` or `header` typed-values, or they can be passed as arguments to methods. List expressions are not l-values. Nested lists are permitted.

For example, the following example uses a list expression to pass multiple header fields simultaneously to a learning provider:

```
extern LearningProvider {
    void learn<T>(in T data);
}
LearningProvider() unit;

unit.learn( { hdr.ethernet.srcAddr, hdr.ipv4.src } );
```

List expressions can be used as initializers to structures; in this case, the list expression must have the same number of elements as the number of fields in the destination structure; structure fields are assigned in order with the values in the list expression:

```
struct S {
    bit<32> a;
    bit<32> b;
}
const S x = { 10, 20 };
```

List expressions can also be used to initialize variables whose type is a `tuple` type:

```
tuple<bit<32>, bool> x = { 10, false };
```

8.11. Set expressions

Some P4 expressions denote sets of values (`set<T>`, for some type `T`; see Section 7.2.7.1). These expressions can appear only in a few contexts.

```
keysetExpression
    : tupleKeysetExpression
    | simpleKeysetExpression
    ;

tupleKeysetExpression
    : '(' simpleKeysetExpression ',' simpleExpressionList ')'
    ;

simpleExpressionList
    : simpleKeysetExpression
    | simpleExpressionList ',' simpleKeysetExpression
    ;

simpleKeysetExpression
    : expression
    | DEFAULT
    | DONTCARE
    | expression MASK expression
    | expression RANGE expression
    ;
```

The mask (`&&&`) and range (`..`) operators have the same precedence, just above `&`.

For example, the `select` expression (Section 11.6) has the following shape:

```
select (expression) {
    set1: state1;
    set2: state2;
   ...
}
```

In this context the expressions `set1, set2`, etc. evaluate to sets of values. The `select` expression tests whether its argument belongs to any of the following sets.

8.11.1. Singleton sets

In a set context a simple expression denotes a set containing a single element. For example:

```
select (hdr.ipv4.version) {
   4: continue;
}
```

The label `4` is a set expression denoting the set consisting of the single value `4`.

8.11.2. The universal set

In a set context the `default` or `_` expressions denote a set containing all possible elements.

```
select (hdr.ipv4.version) {
   4: continue;
   _: reject;
}
```

8.11.3. Cubes

The mask `&&&` infix operator takes two arguments of the same `bit<W>` type, and creates a value of type `set<bit<W>>`. The right value is a "mask", where each 0 bit in the mask indicates a "don't care" bit. The set denoted by `a &&& b` is defined as

```
a &&& b = { c of type bit<W> where a & b = c & b }
```

(This set looks like a cube in the Cartesian space `{0, 1}^W`.) For example:

```
8w0x0A &&& 8w0x0F
```

denotes a set that contains 16 different 8-bit values, whose bit-pattern is `XXXX1010`, where the value of an `X` can be any bit. Note that there may be multiple ways to express a keyset using a mask operator; for example, `8w0xFA &&& 8w0x0F` denotes the same keyset as in the example above.

P4 targets may impose additional restrictions on the expressions on the left and right-hand side of a mask operator: for example, they may require that either or both positions be compile-time known values.

8.11.4. Ranges

The range `..` infix operator takes two arguments of the same type `T bit<W>` or `int<W>` and creates a value of type `set<T>`. The set contains all values numerically between the first and the second, inclusively. For example:

```
4w5 .. 4w8
```

denotes a set with values `4w5, 4w6, 4w7`, and `4w8`.

8.11.5. Tuples of sets

Multiple set expressions can be grouped in a tuple using parentheses:

```
select(hdr.ipv4.ihl, hdr.ipv4.protocol) {
     (4w0x5, 8w0x1): parse_icmp;
     (4w0x5, 8w0x6): parse_tcp;
     (4w0x5, 8w0x11): parse_udp;
     (_, _): accept; }
```

The type of a tuple of sets is a set of tuples.

8.12. Operations on struct types

The only operation defined on values with a structure type is member access operation, indicated using the dot (".") operator (e.g., `s.field`). Field extraction from an l-value produces an l-value. Structs can also be copied using assignment; this is only possible between structs that have the same type.

A struct object can be initialized with a list expression, as described in Section 8.10.

8.13. Operations on headers

Headers provide the same operations as structs. Assignment between headers also copies the "validity" header bit.

The method `isValid()` returns the value of the header's "validity" bit.

The method `setValid()` sets the header's validity bit to "true". It can only be applied to an l-value.

The method `setInvalid()` sets the header's validity bit to "false". It can only be applied to an l-value.

The result of reading or writing a field in an invalid header is undefined. The result of reading an uninitialized header field is undefined — even if the header itself is valid.

A header object can be initialized with a list expression, similar to a struct — the list fields are assigned to the header fields in the order they appear. In this case the header automatically becomes valid:

```
header H { bit<32> x; bit<32> y; }
H h;
h = { 10, 12 };  // This also makes the header h valid
```

8.14. Operations on header stacks

A header stack is a fixed-size array of headers with the same type. The valid elements of a header stack need not be contiguous. P4 provides a set of computations for manipulating header stacks. A header stack `hs` declared as `h[n]` can be understood in terms of the following pseudocode:

```
// type declaration
struct hs_t {
  bit<32> nextIndex;
  bit<32> size;
  h[n] data;  // Ordinary array
}

// instance declaration and initialization
hs_t hs;
```

```
hs.nextIndex = 0;
hs.size = n;
```

Intuitively, a header stack can be thought of as a struct containing an ordinary array of headers `hs` and a counter `nextIndex` that can be used to simplify the construction of parsers for header stacks, as discussed below. The `nextIndex` counter is initialized to `0`.

Given a header stack value `hs` of size `n`, the following expressions are legal:

- `hs[index]`: result is a reference to the header at the specified position within the stack; if `hs` is an l-value, the result is also an l-value. The header may be invalid. Some targets may impose the constraint that the index expression evaluates to a compile-time known value. Accessing a header stack `hs` with an index less than `0` or greater than `hs.size` results in an undefined value.
- `hs.size`: result is a 32-bit unsigned integer that returns the size of the header stack (a compile-time constant).
- assignment from a header stack `hs` into another stack requires the two stacks to have the same types and sizes. All components of `hs` are copied, including its elements and their validity bits, as well as `nextIndex`.

To help programmers write parsers for header stacks, P4 also offers computations that automatically advance through the stack as elements are parsed:

- `hs.next`: result is a reference to the element with index `hs.nextIndex` in the stack. May only be used in a `parser`. If the stack's `nextIndex` counter is greater than or equal to `size`, then evaluating this expression results in a transition to the `reject` state and sets the error to `error.StackOutOfBounds`. If `hs` is an l-value, then `hs.next` is also an l-value.
- `hs.last`: result is a reference to the element with index `hs.nextIndex - 1` in the stack, if such an element exists. May only be used in a `parser`. If the `nextIndex` counter is less than `1`, or greater than `size`, then evaluating this expression results in a transition to the `reject` state and sets the error to `error.StackOutOfBounds`. Unlike `hs.next`, the resulting reference is never an l-value.
- `hs.lastIndex`: result is an `int<32>` that encodes the index `hs.nextIndex - 1`. May only be used in a `parser`. If the `nextIndex` counter is `0`, then evaluating this expression produces an undefined value.

Finally, P4 offers the following computations that can be used to manipulate the elements at the front and back of the stack:

- `hs.push_front(int count)`: shift "right" by `count`. The first `count` elements become invalid. The last `count` elements in the stack are discarded. The `hs.nextIndex` counter is incremented by `count`. The `count` argument must be a positive integer that is a compile-time known value. The return type is `void`.
- `hs.pop_front(int count)`: shift "left" by `count` (i.e., element with index `count` is copied in stack at index `0`). The last `count` elements become invalid. The `hs.nextIndex` counter is decremented by `count`. The `count` argument must be a positive integer that is a compile-time known value. The return type is `void`.

8.15. Operations on Header Unions

A variable declared with a union type is initially invalid:

```
header H1 {
  bit<8> f;
}

header H2 {
  bit<16> g;
}

header_union U {
  H1 h1;
  H2 h2;
}

U u; // u invalid
```

This also implies that each of the headers `h1` through `hn` contained in it is also invalid. Unlike headers, a union cannot be initialized.

However, we can change the validity of a header union by assigning a valid header to one of its elements:

```
header H1
U u;
H1 my_h1 = { 8w0 }; // my_h1 is valid
u.h1 = my_h1;       // u and u.h1 are both valid
```

We can also assign a list to an element of a header union,

```
U u;
u.h2 = { 16w1 };     // u and u.h1 are valid
```

or set their validity bits directly.

```
U u;
u.h1.setValid();     // u and u.h1 are valid
H1 my_h1 = u.h1;         // my_h1 contains an undefined value
~End P4Exampl
Note that reading an uninitialized header produces an undefined value, even if the header is itself

More formally, if `u` is an expression whose type is a header union
`U` with fields ranged over by `hi`, then we can use the following
operations to manipulate the union value:

- `u.hi.setValid():` sets the valid bit for header `hi` to true and sets
  the valid bit for all other headers to false (which implies that
  reading these headers will return an unspecified value).

- `u.hi.setInvalid()`: if the valid bit for `hi` is true then sets it to
  false (which implies that reading u.hi will return an unspecified
  value); if the valid bit for `hi` is already false, then this
  expression has no effect. In particular, if `u.hj` was valid for some `hj` other than `hi` before

We can understand an assignment to a union
~Begin P4Example
```

```
u.hi = e
```

as equivalent to

```
u.hi.setValid();
u.hi = e;
```

if `e` is valid and

```
u.hi.setInvalid();
```

otherwise.

Supplying an expression with a union type to `emit` simply emits the single header that is valid, if any.

The following example shows how we can use header unions to represent IPv4 and IPv6 headers uniformly:

```
header_union IP {
    IPv4 ipv4;
    IPv6 ipv6;
}
struct Parsed_packet {
   Ethernet ethernet;
   IP ip;
}
parser top(packet_in b, out Parsed_packet p) {
    state start {
       b.extract(p.ethernet);
       transition select(p.ethernet.etherType) {
           16w0x0800 : parse_ipv4;
           16w0x86DD : parse_ipv6;
       }
   }
   state parse_ipv4 {
       b.extract(p.ip.ipv4);
       transition accept;
   }
   state parse_ipv6 {
       b.extract(p.ip.ipv6);
       transition accept;
   }
}
```

As another example, we can also use unions to parse (selected) TCP options:

```
header Tcp_option_end_h {
    bit<8> kind;
}
header Tcp_option_nop_h {
    bit<8> kind;
}
header Tcp_option_ss_h {
    bit<8>  kind;
    bit<32> maxSegmentSize;
}
header Tcp_option_s_h {
    bit<8>  kind;
```

```
    bit<24> scale;
}
header Tcp_option_sack_h {
    bit<8>         kind;
    bit<8>         length;
    varbit<256>    sack;
}
header_union Tcp_option_h {
    Tcp_option_end_h  end;
    Tcp_option_nop_h  nop;
    Tcp_option_ss_h   ss;
    Tcp_option_s_h    s;
    Tcp_option_sack_h sack;
}

typedef Tcp_option_h[10] Tcp_option_stack;

struct Tcp_option_sack_top {
    bit<8> kind;
    bit<8> length;
}

parser Tcp_option_parser(packet_in b, out Tcp_option_stack vec) {
    state start {
        transition select(b.lookahead<bit<8>>()) {
            8w0x0 : parse_tcp_option_end;
            8w0x1 : parse_tcp_option_nop;
            8w0x2 : parse_tcp_option_ss;
            8w0x3 : parse_tcp_option_s;
            8w0x5 : parse_tcp_option_sack;
        }
    }
    state parse_tcp_option_end {
        b.extract(vec.next.end);
        transition accept;
    }
    state parse_tcp_option_nop {
         b.extract(vec.next.nop);
         transition start;
    }
    state parse_tcp_option_ss {
         b.extract(vec.next.ss);
         transition start;
    }
    state parse_tcp_option_s {
         b.extract(vec.next.s);
         transition start;
    }
    state parse_tcp_option_sack {
         bit<32> n = b.lookahead<Tcp_option_sack_top>().length;
         b.extract(vec.next.sack, n);
         transition start;
```

```
    }
}
```

8.16. Function calls, method invocations

Functions can be invoked using the function call syntax.

```
expression
    : ...
    | expression '<' typeArgumentList '>' '(' argumentList ')'
    | expression '(' argumentList ')'

argumentList
    : /* empty */
    | nonEmptyArgList
    ;

nonEmptyArgList
    : argument
    | nonEmptyArgList ',' argument
    ;

argument
    : expression
    ;

typeArgumentList
    : typeRef
    | typeArgumentList ',' typeRef
    ;
```

Function arguments are evaluated in order, left to right, before the function invocation takes place. The calling convention is copy-in/copy-out (Section 8.6). For generic functions the type arguments can be explicitly specified in the function call. No implicit casting is used for function arguments; the types of the arguments must match the parameter types exactly.

Similar to the C programming language, the result returned by a function call is discarded when the function call is used as a statement.

8.17. Constructor invocations

Several P4 constructs denote resources that are allocated at compilation time:

- `extern` objects
- `parser`s
- `control` blocks
- `package`s

Allocation of such objects can be performed in two ways:

- using constructor invocations, which are expressions that return an object of the corresponding type.
- using instantiations, described in Section 9.3. (Instantiations are similar to constant declarations.)

The syntax of a constructor invocation is similar to a function call. Constructors are evaluated entirely at compilation-time (see Section 16). In consequence, all constructor arguments must also be expressions that can be evaluated at compilation time.

The following example shows a constructor invocation for setting the target-dependent implementation property of a table:

```
extern ActionProfile {
    ActionProfile(bit<32> size);  // constructor
}
table tbl {
    actions = { ... }
    implementation = ActionProfile(1024);  // constructor invocation
}
```

9. Constants and variable declarations

9.1. Constants

Constant values are defined with the syntax:

```
constantDeclaration
    : optAnnotations CONST typeRef name '=' initializer ';'
    ;

initializer
    : expression
    ;
```

This introduces a constant whose value has the specified type. The following are all legal constant declarations:

```
const bit<32> COUNTER = 32w0x0;
struct Version {
    bit<32> major;
    bit<32> minor;
}
const Version version = { 32w0, 32w0 };
```

`initializer` must be a compile-time known value.

9.2. Variables

Local variables can be declared using variable declarations:

```
variableDeclaration
    : annotations typeRef name optInitializer ';'
    | typeRef name optInitializer ';'
    ;

optInitializer
    : /* empty */
    | '=' initializer
    ;
```

Variables without an initializer are uninitialized (except for header stacks, which have their `nextIndex` counter initialized to 0, as discussed in 8.14). The language places no restriction on the types of the variables: most P4 types that can be written explicitly can be used (e.g., base types, `struct`, `header`, header stack, `tuple`). One cannot declare variables with types that are only synthesized by the compiler (e.g., `set`), or with `parser`, `control`, `package`, or `extern` types. Objects of the latter types must be declared using instantiations (see Section 9.3).

Reading the value of a variable that has not been initialized provides an undefined result. The compiler should attempt to detect and flag such reads statically.

Variables declarations can appear in the following places in a P4 program:

- In a block statement
- In a `parser` state
- In an `action` body
- In a `control` block apply block
- In the list of local declarations in a `parser`
- In the list of local declarations in a `control`

Variables are local, and behave like stack-allocated variables from languages such as C. The value of a variable is never preserved from one invocation of its enclosing block to the next, so variables cannot be used to maintain state between different network packets.

9.3. Instantiations

Instantiations are similar to variable declarations, but they are reserved for the types with constructors (`extern` objects, `control` blocks, `parser`s and `package`s):

```
instantiation
    : typeRef '(' argumentList ')' name ';'
    | annotations typeRef '(' argumentList ')' name ';'
    ;
```

An instantiation looks like a constructor invocation followed by a name. Instantiations are always executed at compilation-time (Section 16.1). The effect is to allocate an object with the specified name, and to bind it to the result of the constructor invocation.

The following example shows how a hypothetical counter bank can be instantiated:

```
// from target library
enum CounterType {
   Packets,
   Bytes,
   Both
}
extern Counter {
    Counter(bit<32> size, CounterType type);
    void increment(in bit<32> index);
}
// user program
control c(...) {
    Counter(32w1024, CounterType.Both) ctr;  // instantiation
    apply { ... }
}
```

10. Statements

Statements (except the block statement) must end with a semicolon, C-style.

Statements can appear in several places:

- Within `parser` states
- Within a `control` block
- Within an `action`

There are restrictions for the kinds of statements that can appear in each of these places. For example, conditionals are not supported in parsers, and `switch` statements are only supported in control blocks. We present here the most general case, for control blocks.

```
statement
    : assignmentOrMethodCallStatement
    | conditionalStatement
    | emptyStatement
    | blockStatement
    | exitStatement
    | returnStatement
    | switchStatement
    ;

assignmentOrMethodCallStatement
    : lvalue '(' argumentList ')' ';'
    | lvalue '<' typeArgumentList '>' '(' argumentList ')' ';'
    | lvalue '='  expression ';'
    ;
```

In addition, parsers support a `transition` statement (Section 11.5).

(Due to limitations in the Bison parser generator we have grouped all productions starting with a l-value in the same rule.)

10.1. Assignment

Assignment is denoted with the = sign.

An assignment evaluates first the expression on the left hand side, which must evaluate to an l-value, then the expression on the right hand side, and copies that value into the left hand side. Derived types (e.g. `structs`) are copied recursively. `header`s are copied, including their "validity" bits. Assignment is not defined for `extern` values.

10.2. The empty statement

The empty statement is a no-op.

```
emptyStatement
    : ';'
    ;
```

10.3. The block statement

A block statement is denoted by curly braces, as in C. It contains a sequence of statements and declarations, which are executed sequentially. The variables, constants and instantiations within a

block statement are only visible within the block statement.

```
blockStatement
    : optAnnotations '{' statOrDeclList '}'
    ;

statOrDeclList
    : /* empty */
    | statOrDeclList statementOrDeclaration
    ;

statementOrDeclaration
    : variableDeclaration
    | constantDeclaration
    | statement
    | instantiation
    ;
```

10.4. The return statement

The `return` statement immediately terminates the execution of the `action` or `control` that contains the `return` statement. `return` statements are not allowed within parsers.

```
returnStatement
    : RETURN ';'
    ;
```

10.5. The exit statement

The `exit` statement immediately terminates the execution of all the blocks currently executing: the current `action` (if invoked within an `action`), the current `control` and all its callers. `exit` statements are not allowed within parsers.

```
exitStatement
    : EXIT ';'
    ;
```

10.6. The conditional statement

The conditional statement is very similar in syntax and semantics with the corresponding C if statement. The only difference is that the only acceptable type for the condition expression in P4 is Boolean (and not integer). The conditional statement cannot be used within a `parser`.

```
conditionalStatement
    : IF '(' expression ')' statement
    | IF '(' expression ')' statement ELSE statement
    ;
```

When using nested `if` statements, the `else` applies to the innermost `if` that does not have an `else` statement.

10.7. The switch statement

The `switch` can only be used within `control` blocks.

```
switchStatement
    : SWITCH '(' expression ')' '{' switchCases '}'
    ;

switchCases
    : /* empty */
    | switchCases switchCase
    ;

switchCase
    : switchLabel ':' blockStatement
    | switchLabel ':'  // fall-through
    ;

switchLabel
    : name
    | DEFAULT
    ;
```

The expression within the `switch` statement is restricted to be the result of a table's invocation (more details are given in Section 12.2.2).

If a switch label is not followed by a block statement it is fall-through to the next label. However, if a block statement is present, there is no fall-through. Note, that this is different from C `switch` statements, where a `break` is needed to prevent fall-through. It is legal to have no matching label for some actions, or no `default` label. No label can appear twice in a switch statement.

```
switch (t.apply().action_run) {
   action1:          // fall-through to action2:
   action2: { ...}
   action3: { ...}  // no fall-through from action2 to action3 labels
}
```

Please note that the `default` label of the `switch` statement is used to match on the kind of action executed, no matter whether there was a table hit or miss. The `default` label does not indicate that the table missed and the `default_action` was executed.

11. Packet parsing in P4

This section describes the P4 constructs specific to parsing network packets.

11.1. Parser states

A P4 parser describes a state-machine with one start state and two final states. The start state is always named `start`. The two final states are named `accept` (indicating successful parsing) and `reject` (indicating a parsing failure). The `start` state is part of the parser, while the `accept` and `reject` states are logically outside of the parser. Figure 8 illustrates the general structure of a parser state-machine.

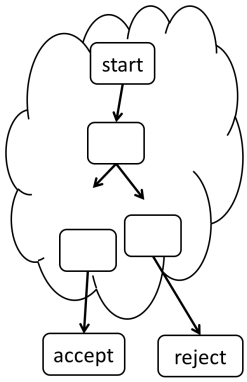

Figure 8. Parser FSM structure.

11.2. Parser declarations

P4 programmers are expected to provide parser declarations for all programmable parsers of a target.

```
parserDeclaration
    : parserTypeDeclaration optConstructorParameters
      '{' parserLocalElements parserStates '}'
    ;

parserLocalElements
    : /* empty */
    | parserLocalElements parserLocalElement
    ;

parserStates
    : parserState
    | parserStates parserState
    ;
```

Parsers cannot be generic (unlike parser type declarations); thus when used in the context of a `parserDeclaration` the `parserTypeDeclaration` cannot contain any type parameters. I.e., the following declaration is illegal:

```
parser P<H>(inout H data) { ... }
```

At least one state, named `start`, must be present in any `parser`. A parser cannot contain two states with the same name. A parser cannot define the `accept` and `reject` states; these are logically outside of the parser.

State declarations are described below. Preceding the parser states, a `parser` may also contain a list of local elements. These can be constants, variables, or instantiations of objects that may be used within the parser. Such objects may be instantiations of `extern` objects, or other `parser`s that may be invoked as subroutines. P4 forbids the instantiation of `control` blocks within a `parser`.

```
parserLocalElement
    : constantDeclaration
    | variableDeclaration
    | instantiation
```

```
    ;
```

For an example containing a complete declaration of a parser see Section 5.3.

For a description of the `optConstructorParameters`, used for building parameterized parsers, see Section 13.

11.3. The Parser abstract machine

We explain the semantics of P4 parsers program using an abstract machine that manipulates a data structure named `ParserModel`. The abstract machine is described using pseudo-code.

A parser starts execution in the `start` state and ends execution when one of the `reject` or `accept` states has been reached.

```
ParserModel {
    error       parseError;
    state       currentState;
    onPacketArrival(packet p) {
        ParserModel.parseError = error.NoError;
        ParserModel.currentState = start;
        execute(ParserModel.currentState);
    }
}
```

11.4. Parser states

A parser state has a name and a body. The body of the state describes data processing performed when the parser transitions to the specified state. This data processing may consist of:

- Data extraction from packet into headers
- Invocations of methods of `extern` blocks (e.g., checksum computations)
- Checks for data validity
- Transition to other states

A state is declared with the following syntax:

```
parserState
    : optAnnotations STATE name
     '{' parserStatements transitionStatement '}'
    ;
```

```
parserStatements
    : /* empty */
    | parserStatements parserStatement
    ;

parserStatement
    : assignmentOrMethodCallStatement
    | variableDeclaration
    | constantDeclaration
    | parserBlockStatement
```

```
    ;

parserBlockStatement
    : optAnnotations '{' parserStatements '}'
    ;
```

The state body contains a sequence of:

- local variable declarations
- assignment statements
- method calls. These can serve multiple purposes:
  - `verify` calls
  - method invocations (e.g., for extracting data out of packets),
  - invocations of other parsers.

The parser state body ends with an optional `transition` statement, which transfers control to the next state. Certain targets may place restrictions on the complexity of the expressions that can be evaluated while executing a parser.

Semantics: The execution of a state entails the sequential execution of the statements in the state body.

11.5. Transition statements

The `transition` statement has the following syntax:

```
transitionStatement
    : /* empty */
    | TRANSITION stateExpression
    ;

stateExpression
    : name ';'
    | selectExpression
    ;
```

The execution of the transition statement causes `stateExpression` to be evaluated, and the flow of control to be transferred to the resulting state.

Semantics: in terms of the ParserModel the above statement is equivalent to:

```
ParserModel.currentState = eval(stateExpression)
```

For example, this statement:

```
transition accept;
```

will terminate the execution of the current parser transitioning to the `accept` state.

If the body of a state block does not end with a `transition` statement, the implied statement is

```
transition reject;
```

11.6. select expressions

A `select` expression evaluates to a state. The syntax is the following:

```
selectExpression
    : SELECT '(' expressionList ')' '{' selectCaseList '}'
    ;

selectCaseList
    : /* empty */
    | selectCaseList selectCase
    ;

selectCase
    : keysetExpression ':' name ';'
    ;
```

If expressionList has type `tuple<T>`, all `keysetExpression` must have type `set<tuple<T>>`.

Semantics: the meaning of the following select expression:

```
select(e) {
    ks[0]: s[0];
    ks[1]: s[1];
    ...
    ks[n-2]: s[n-1];
    _ : sd;  // ks[n-1] is default
}
```

is defined in pseudo-code as:

```
key = eval(e);
for (int i=0; i < n; i++) {
    keyset = eval(ks[i]);
    if (keyset.contains(key)) return s[i];
}
verify(false, error.NoMatch);
```

Some targets may require that all keyset expressions in a select expression must be compile-time known values. Keysets are evaluated in order, from top to bottom as implied by the pseudo-code above; the first keyset that includes the value in the `select` argument provides the result state. If no label matches, the execution triggers a runtime error with the standard error code `error.NoMatch`.

Note that this implies that all cases after a `default` or `_` label are unreachable; the compiler should emit warnings about this case. This constitutes an important difference between `select` expression and `switch` statement in C language, where the order does not matter; the keysets of a `select` expression may "overlap").

The most typical case for using `select` expressions is to compare the value of a field from a recently-extracted header against a set of constant values, as in the following example:

```
header IPv4_h { ... bit<8> protocol; ... }
struct P { ... IPv4_h ipv4; ... }
P headers;
select (headers.ipv4.protocol) {
    8w6  : parse_tcp;
    8w17 : parse_udp;
    _    : accept;
}
```

For example, to detect TCP reserved ports (< 1024) one could write:

```
select (p.tcp.port) {
    16w0 &&& 16w0xFC00: well_known_port; // top 6 bits are zero
    _: other_port;
}
```

The expression `16w0 &&& 16w0xFC00` describes the set of 16-bit values that have their top 6 bits zero.

11.7. verify

The `verify` statement provides a simple form of error handling. `verify` can only be invoked within a parser; it is used syntactically as if it were a function with the following signature:

```
extern void verify(in bool condition, in error err);
```

If the first argument is `true`, the execution of the statement has no side-effects. If the first argument is `false`, it causes an immediate transition to the `reject` state, which causes immediate parsing termination; at the same time, the `parserError` associated with the parser is set to the value of the second argument.

Semantics: in terms of the `ParserModel` the semantics of a `verify` statement is given by:

```
ParserModel.verify(bool condition, error err) {
    if (condition == false) {
        ParserModel.parserError = err;
        ParserModel.currentState = reject;
    }
}
```

11.8. Data extraction from packets

The P4 core library contains the following declaration of a built-in `extern` type called `packet_in` that represents incoming network packets. The `packet_in` extern is special: it cannot be instantiated by the user explicitly. Instead, the architecture supplies a separate instance for each `packet_in` argument to a top-level `parser` or `control` block.

```
extern packet_in {
    void extract<T>(out T headerLvalue);
    void extract<T>(out T variableSizeHeader, in bit<32> varFieldSizeBits);
    T lookahead<T>();
    bit<32> length();  // This method may be unavailable in some architectures
    void advance(bit<32> bits);
}
```

To extract data from a packet represented by an argument `b` with type `packet_in`, a parser invokes the `extract` methods of `b`. There are two variants of the `extract` method: a one-argument variant for extracting fixed-size headers, and a two-argument variant for extracting variable-sized headers. Because these operations can cause runtime verification failures (see below), these methods can only be executed within parsers.

When extracting data into a bit-string or integer, the first packet bit is extracted to the most significant bit of the integer.

Some targets may perform cut-through packet processing, i.e., they may start processing a packet before its length is known (i.e., before all bytes have been received). On such a target calls to the `packet_in.length()` method cannot be implemented. Attempts to call this method should be flagged as errors (either at compilation time by the compiler back-end, or when attempting to load the compiled P4 program onto a target that does not support this method).

Semantics: we describe the semantics of these operations in terms of the following abstract model of a packet data structure (pseudo-code):

```
packet_in {
    unsigned nextBitIndex;
    byte[] data;
    unsigned lengthInBits;
    void initialize(byte[] data) {
        this.data = data;
        this.nextBitIndex = 0;
        this.lengthInBits = data.sizeInBytes * 8;
    }
    bit<32> length() { return this.lengthInBits / 8; }
}
```

11.8.1. extract — single argument

The single-argument extract method has the following P4 declaration:

```
void extract<T>(out T headerLeftValue);
```

`headerLeftValue` is an expression that should evaluate to a l-value (see Section 6.6) of type `header` with a fixed width. If this method executes successfully, on completion the `headerLvalue` is filled with data from the packet and its validity bit is set. This method may fail by executing a failed `verify` call (e.g., not enough bits left in packet to fill the specified header). For example, to extract an Ethernet header one can invoke:

```
struct Result { ... Ethernet_h ethernet; ... }
parser P(packet_in b, out Result r) {
    state start {
        b.extract(r.ethernet);
    }
}
```

Semantics: the semantics of extract is given in terms of the following pseudo-code method of the `packet` class shown above. We use the special `valid$` identifier to indicate the hidden valid bit of a header, `isNext$` to indicate that the l-value was obtained using `next`, and `nextIndex$` to indicate the corresponding header stack properties.

```
void packet_in.extract<T>(out T headerLValue) {
   bitsToExtract = sizeofInBits(headerLValue);
   lastBitNeeded = this.nextBitIndex + bitsToExtract;
   ParserModel.verify(this.lengthInBits >= lastBitNeeded, error.PacketTooShort);
   headerLValue = this.data.extractBits(this.nextBitIndex, bitsToExtract);
   headerLValue.valid$ = true;
   if headerLValue.isNext$ {
     verify(headerLValue.nextIndex$ < headerLValue.size, error.StackOutOfBounds);
     headerLValue.nextIndex$ = headerLValue.nextIndex$ + 1;
   }
   this.nextBitIndex += bitsToExtract;
}
```

11.8.2. extract — two arguments

The two-argument extract method has the following declaration:

```
void extract<T>(out T headerLvalue, in bit<32> variableFieldSize);
```

`headerLvalue` must be a l-value representing a header that contains *exactly one* `varbit` field. `variableFieldSize` is an expression evaluating to a `bit<32>` value which indicates the number of bits to be extracted into the unique `varbit` field of the header (this size is not the size of the complete header, just the size of the `varbit` field).

Semantics: the semantics of a two-argument extract invocation is given in terms of the following pseudo-code:

```
void packet_in.extract<T>(out T headerLvalue,
                          in bit<32> variableFieldSize) {
   bitsToExtract = sizeOfFixedPart(headerLvalue) + variableFieldSize;
   lastBitNeeded = this.nextBitIndex + bitsToExtract;
   ParserModel.verify(this.lengthInBits >= lastBitNeeded, error.PacketTooShort);
   ParserModel.verify(bitsToExtract <= headerLvalue.maxSize, error.HeaderTooShort);
   headerLvalue = this.data.extractBits(this.nextBitIndex, bitsToExtract);
   headerLvalue.varbitField.size = variableFieldSize;
   headerLvalue.valid$ = true;
   if headerLValue.isNext$ {
     verify(headerLValue.nextIndex$ < headerLValue.size, error.StackOutOfBounds);
     headerLValue.nextIndex$ = headerLValue.nextIndex$ + 1;
   }
   this.nextBitIndex += bitsToExtract;
}
```

The example below shows one way that IPv4 options can be extracted by splitting the IPv4 header into two separate headers:

```
// IPv4 header without options
header IPv4_no_options_h {
   bit<4>   version;
   bit<4>   ihl;
   bit<8>   diffserv;
   bit<16>  totalLen;
   bit<16>  identification;
   bit<3>   flags;
   bit<13>  fragOffset;
   bit<8>   ttl;
   bit<8>   protocol;
   bit<16>  hdrChecksum;
   bit<32>  srcAddr;
   bit<32>  dstAddr;
}

header IPv4_options_h {
   varbit<320> options;
}

struct Parsed_headers {
    ...
    IPv4_no_options_h ipv4;
```

```
    IPv4_options_h    ipv4options;
}

error { InvalidIPv4Header }

parser Top(packet_in b, out Parsed_headers headers) {
   ...
   state parse_ipv4 {
       b.extract(headers.ipv4);
       verify(headers.ipv4.ihl >= 5, error.InvalidIPv4Header);
       transition select (headers.ipv4.ihl) {
           5: dispatch_on_protocol;
           _: parse_ipv4_options;
   }

   state parse_ipv4_options {
       b.extract(headers.ipv4options,
                 (bit<32>)(((bit<16>)headers.ipv4.ihl - 5) * 32));
       transition dispatch_on_protocol;
   }
}
```

11.8.3. Lookahead

`lookahead` is a method provided by the `packet_in` packet abstraction that evaluates to a set of bits from the input packet without advancing the `nextBitIndex` pointer. Similar to `extract`, it will transition to `reject` and set the error if there are not enough bits in the packet. One invokes `lookahead` as follows:

```
b.lookahead<T>()
```

where `T` must be a type with fixed width. In case of success the result of the evaluation of `lookahead` returns a value of type `T`.

Semantics: in terms of the abstract model the semantics of lookahead is given by the following pseudo-code:

```
T packet_in.lookahead<T>() {
   bitsToExtract = sizeof(T);
   lastBitNeeded = this.nextBitIndex + bitsToExtract;
   ParserModel.verify(this.lengthInBits >= lastBitNeeded, error.PacketTooShort);
   T tmp = this.data.extractBits(this.nextBitIndex, bitsToExtract);
   return tmp;
}
```

Examples:

```
header Tcp_option_sack_top { ...}
state start {
    transition select(b.lookahead<bit<8>>()) {
        0: parse_tcp_option_end;
        1: parse_tcp_option_nop;
        2: parse_tcp_option_ss;
        3: parse_tcp_option_s;
        5: parse_tcp_option_sack;
```

```
    }
}
state parse_tcp_option_sack {
     b.extract(vec.next.sack,
               (bit<32>)(b.lookahead<Tcp_option_sack_top>().length));
     transition next;
}
```

11.8.4. Skipping bits

There are two ways to skip over bits in an input packet without assigning them to a header:

One can extract to the underscore identifier, by specifying explicitly the type of the data:

```
b.extract<T>(_)
```

Alternatively, one can use the `advance` method of the packet when the number of bits to skip is known. The meaning of this method is given in pseudo-code as:

```
void packet_in.advance(bit<32> bits) {
   lastBitNeeded = this.nextBitIndex + bits;
   ParserModel.verify(this.lengthInBits >= lastBitNeeded, error.PacketTooShort);
   this.nextBitIndex += bits;
}
```

11.9. Parsing header stacks

During parsing, a header stack provides two properties: `next` and `last`. Consider the following declaration, which defines a stack for representing the headers of a packet with at most 10 MPLS headers:

```
header Mpls_h {
    bit<20> label;
    bit<3>  tc;
    bit     bos;
    bit<8>  ttl;
}
Mpls_h[10] mpls;
```

The expression `mpls.next` represents an l-value of type `Mpls_h` that references an element in the `mpls` stack. Initially, `mpls.next` refers to the first element of stack. It is automatically advanced on each successful call to `extract`. The `mpls.last` property refers to the element immediately preceding `next` if such an element exists. Attempting to access `mpls.next` element when the stack's `nextIndex` counter is greater than or equal to `size` causes a transition to `reject` and sets the error to `error.StackOutOfBounds`. Likewise, attempting to access `mpls.last` when the `nextIndex` counter is equal to 0 causes a transition to the `reject` state and sets the error to `error.StackOutOfBounds`.

The following example shows a simplified parser for MPLS processing:

```
struct Pkthdr {
   Ethernet_h ethernet;
   Mpls_h[3] mpls;
   // other headers omitted
}
parser P(packet_in b, out Pkthdr p) {
    state start {
        b.extract(p.ethernet);
```

```
        transition select(p.ethernet.etherType) {
           0x8847: parse_mpls;
           0x0800: parse_ipv4;
        }
    }
    state parse_mpls {
         b.extract(p.mpls.next);
         transition select(p.mpls.last.bos) {
            0: parse_mpls; // This creates a loop
            1: parse_ipv4;
         }
    }
    // other states omitted
}
```

11.10. Invoking sub-parsers

P4 allows parsers to invoke the services of other parsers, similar to subroutines. To invoke the services of another parser, the sub-parser must be first instantiated; the services of an instance are invoked by calling it using its apply method.

The following example shows a sub-parser invocation:

```
parser callee(packet_in packet, out IPv4 ipv4) { ...}
parser caller(packet_in packet, out Headers h) {
     callee() subparser;  // instance of callee
     state subroutine {
          subparser.apply(packet, h.ipv4);  // invoke sub-parser
     }
}
```

Semantics: the semantics of a subparser invocation can be described as follows:

- The state invoking the sub-parser is split into two half-states at the parser invocation statement.
- The top half includes a transition to the sub-parser `start` state.
- The sub-parser's `accept` state is identified with the bottom half of the current state
- The sub-parser's `reject` state is identified with the reject state of the current parser.

Figure 9 shows a diagram of this process.

Since P4 requires declarations to precede uses, it is impossible to create recursive (or mutually recursive) parsers.

Various targets may impose (static or dynamic) constraints on the number of parser states that can be traversed for processing each packet. For example, a specific compiler implementation may reject parsers where loops cannot be unrolled at compilation time, or it may reject parser cycles that do not advance the cursor within the parsed packet. If a parser aborts execution dynamically because it exceeded the maximum time budget allocated, the parser error should be set to the standard error `error.ParserTimeout`.

12. Control blocks

P4 parsers are responsible for extracting bits from a packet into headers. The headers can be manipulated and transformed within `control` blocks. Control blocks are a P4 construct that are used for describing match-action pipelines. The body of a control block resembles a traditional C

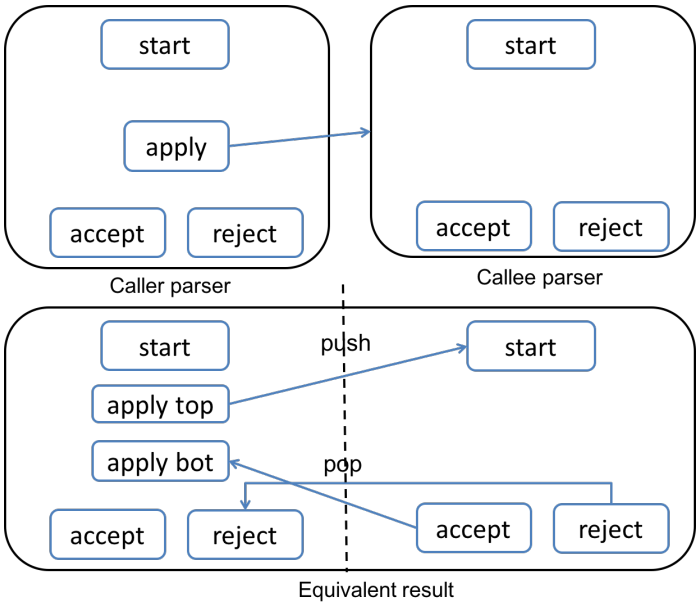

Figure 9. Semantics of invoking a sub-parser: top: original program, bottom: equivalent program.

program. Within the body of a control block, match-action units can be invoked to perform data transformations. Match-action units are represented in P4 by constructs called `tables`.

There is no exceptional control-flow in a `control` block: no equivalent of the `verify` parser statement or of the `reject` state. Error handling has to be performed explicitly by users.

P4 forbids the instantiation of `parser` instances within a `control` block.

A `control` block may start with declarations of `actions`, `tables`, constants, variables and instantiations.

```
controlDeclaration
    : controlTypeDeclaration optConstructorParameters
      /* controlTypeDeclaration cannot contain type parameters */
      '{' controlLocalDeclarations APPLY controlBody '}'
    ;

controlLocalDeclarations
    : /* empty */
    | controlLocalDeclarations controlLocalDeclaration
    ;

controlLocalDeclaration
    : constantDeclaration
    | variableDeclaration
    | actionDeclaration
    | tableDeclaration
    | instantiation
    ;

controlBody
    : blockStatement
```

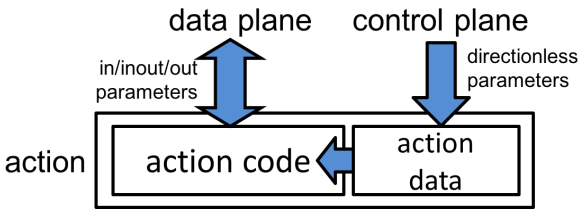

Figure 10. Actions contain code and data. The code is in the P4 program, while the data is set by the control plane. Parameters are bound by the data plane.

```
    ;
```

Controls cannot be generic (unlike control type declarations); thus when used in the context of a `controlDeclaration` the `controlTypeDeclaration` cannot contain any type parameters. I.e., the following declaration is illegal:

```
control C<H>(inout H data) { ... }
```

For a description of the `optConstructorParameters`, which can be used to build parameterized control blocks, see Section 13.

We start by describing the core components of a `control` block, starting with actions.

12.1. Actions

Actions are code fragments that can read and write the data being processed. Actions may contain data values that can be written by the control plane and read by the data plane. Actions are the fundamental construct by which the control-plane can influence dynamically the behavior of the data plane. Figure 10 shows the abstract model of an `action`. Formally, actions are function objects https://en.wikipedia.org/wiki/Function_object.

```
actionDeclaration
    : optAnnotations ACTION name '(' parameterList ')' blockStatement
    ;
```

Syntactically actions resemble functions with no return values. Actions may be declared within a control block; in this case they can only be used within an instance of that control block.

The following example shows an action declaration:

```
action Forward_a(out bit<9> outputPort, bit<9> port) {
    outputPort = port;
}
```

Action parameters may not have `extern` types. Action parameters that have no direction (e.g., `port` in the previous example) indicate action data. All such parameters must be at the end of the parameter list. For actions that appear in a table actions list (described in Section 12.2.1.2), these parameters are bound by the control plane.

The body of an action consists of a sequence of statements and declarations. No switch statements are allowed within an action (the grammar as written permits them, but a semantic check should reject them). Some targets may impose additional restrictions on action bodies—e.g., only straight-line code, with no `if` statements or conditional expressions (`?:`).

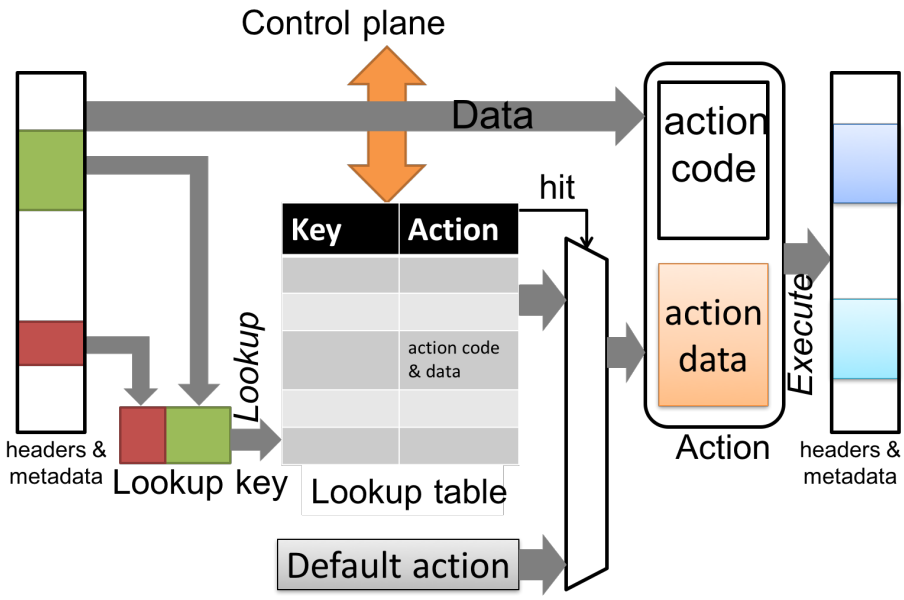

Figure 11. Match-Action Unit Dataflow.

12.1.1. Invoking actions

Actions can be executed in two ways:

- Actions can be implicitly invoked by tables during match-action processing.
- Actions can be explicitly invoked, either from a `control` block or from another `action`. In this case, values for all action parameters must be supplied explicitly, including values for the directionless parameters. The directionless parameters in this case behave like `in` parameters.

12.2. Tables

The P4 `table` construct describes a match-action unit. The structure of a match-action unit is shown in Figure 11. The match-action processing consists of the following steps:

- Key construction.
- Key lookup in a lookup table (the "match" step). The result of key lookup is an "action".
- Action execution (the "action step") over the input data, resulting in mutations of the data.

A `table` declaration introduces a table instance. If one desires to instantiate a table multiple times one needs to declare a table in a separate control block and instantiate that control block multiple times.

The look-up table is a finite map whose contents are manipulated asynchronously (read/write) by the target control-plane, through a separate control-plane API (see Figure 11). Note that the term "table" is overloaded: it can refer to the P4 `table` objects that appear in P4 programs, as well as the internal look-up tables used in targets. In this document, we use the term "match-action unit" whenever possible instead of "table".

Syntactically a table is described by a set of key-value properties. Some of these properties are "standard" properties, but the set of properties can be extended by target-specific compilers as needed.

```
tableDeclaration
    : optAnnotations TABLE name '{' tablePropertyList '}'
    ;

tablePropertyList
    : tableProperty
    | tablePropertyList tableProperty
    ;

tableProperty
    : KEY '=' '{' keyElementList '}'
    | ACTIONS '=' '{' actionList '}'
    | optAnnotations CONST IDENTIFIER '=' initializer ';'
    | optAnnotations IDENTIFIER '=' initializer ';'
    ;
```

The standard table properties include:

- `key`: An expression that describes how the key used for look-up is computed.
- `actions`: A list of all actions that may be found in the table.

In addition, the tables may optionally define the following property,

- `default_action`: an action to execute when the lookup in the lookup table fails to find a match for the key used.

as well as architecture-specific properties (see Sec. 12.2.1.5).

We discuss each of these properties in detail.

A property marked as `const` cannot be changed dynamically by the control-plane. The `key` and `actions` properties are always constant, so the `const` keyword is not needed for these.

12.2.1. Table properties

12.2.1.1. Table keys The `key` is a table property which specifies the key used when looking up an action in the lookup table. The key is given by a set of pairs (expression : matchKind):

```
keyElementList
    : /* empty */
    | keyElementList keyElement
    ;

keyElement
    : expression ':' name optAnnotations ';'
    ;
```

Each name in a `keyElement` must be a constant of type `match_kind` (see Section 7.1.3).

For example, let us consider the following `table` declaration fragment:

```
table Fwd {
    key = {
       ipv4header.dstAddress : ternary;
       ipv4header.version    : exact;
    }
```

```
    ...
}
```

In this example the lookup key is composed of two fields of the `ipv4header` structure: `dstAddress` and `version`. The `match_kind` information attached to each key expression is used for two purposes:

- It is used to synthesize the control-plane API that is used to populate the table. The control-plane API specification is part of a separate document.
- It is used by the compiler back-end to allocate resources for the table's implementation.

The P4 core library contains three predefined `match_kind` identifiers:

```
match_kind {
   exact,
   ternary,
   lpm
}
```

These identifiers correspond to the $P4_{14}$ match kinds with the same names. The semantics of these annotations is actually irrelevant for describing the behavior of the P4 abstract machine; how they are used influences only the control-plane API and the actual implementation of the look-up table. From the point of view of the P4 program, a look-up table is an abstract finite map that is given a key and produces as a result either an action or a "miss" indication, as described in Section 12.2.3.

If a table has no `key` property, then it contains no look-up table, just a default action, which is always executed (i.e., the associated lookup table is always the empty map).

Each key element can have an optional `@name` annotation which is used to synthesize the control-plane visible name for the key field.

12.2.1.2. The list of actions A table must declare all possible actions that may appear within the associated lookup table or in the default action. This is done with the `actions` property; the value of this property is always an `actionList`:

```
actionList
    : actionRef ';'
    | actionList actionRef ';'
    ;

actionRef
    : optAnnotations name
    | optAnnotations name '(' argumentList ')'
    ;
```

Let us consider an example from the Very Simple Switch program in Section 5.3:

```
action Drop_action()
{ outCtrl.outputPort = DROP_PORT; }

action Rewrite_smac(EthernetAddress sourceMac)
{ headers.ethernet.srcAddr = sourceMac; }

table smac {
    key = { outCtrl.outputPort : exact; }
    actions = {
        Drop_action;
        Rewrite_smac;
```

```
    }
}
```

- The `smac table` can contain two types of actions, named `Drop_action` and `Rewrite_mac`.
- The `Rewrite_smac` action has one parameter, which is bound by the control plane.

All actions in the list of actions of a table must have different names; e.g., the following program fragment is illegal:

```
action a() {}
control c() {
    action a() {}
    // Illegal table: two actions with the same name
    table t { actions = { a; .a; } }
}
```

Each action parameter that has a direction (`in`, `inout` or `out`) must be bound in the `actions` list specification; conversely, no directionless parameters may be bound in the list. The expressions supplied as arguments to an `action` are not evaluated until the action is invoked.

```
action a(in bit<32> x) { ...}
bit<32> z;
action b(inout bit<32> x, bit<8> data) { ...}
table t {
    actions = {
       // a; - illegal, x parameter must be bound
       a(5);  // binding a's parameter x to 5
       b(z);  // binding b's parameter x to z
       // b(z, 3);  // - illegal, cannot bind directionless data parameter
       // b(); - illegal, x parameter must be bound
    }
}
```

12.2.1.3. The default action The default action of a table is an action that is invoked automatically by the match-action unit whenever the lookup table does not find a match for the supplied key.

If present, the `default_action` property *must* appear after the `action` property. It may be declared as `const`, indicating that it cannot be changed dynamically by the control-plane. The `default action` *must* be one of the actions that appear in the actions list. In particular, the expressions passed as `in`, `out` or `inout` parameters must be syntactically identical to the expressions used in one of the elements of the `actions` list.

For example, in the above `table` we could set the default action as follows (marking it also as constant — i.e., not changeable by the control plane):

```
const default_action = Rewrite_smac(48w0xAA_BB_CC_DD_EE_FF);
```

Note that the specified default action must supply arguments for the control-plane bound parameters (i.e., the directionless parameters), since the action is synthesized at compilation time. The expressions supplied as arguments for parameters with a direction (`in`, `inout` or `out`) are evaluated when the action is invoked while the expressions supplied as arguments for directionless parameters are evaluated at compile time.

Continuing the example in the previous section, here are various legal and illegal specifications for `table t` above:

```
  default_action = a(5); // OK - no control-plane parameters
  // default_action = a(z); - illegal, a's x parameter is already bound to 5
```

```
default_action = b(z,8w8); // OK - bind b's data parameter to 8w8
// default_action = b(z); - illegal, b's data parameter is not bound
```

If a table does not specify the `default_action` property and no entry matches a given packet, then the table does not affect the packet and processing continues according to the imperative control flow of the program.

12.2.1.4. Table entries A table may be initialized at compile-time with a set of entries. While it is typical that entries are installed in tables by the contol plane program, there are situations in which tables are used to implement fixed algorithms. In such cases, defining the entries that enable the execution of the algorithm in the P4 source, allows the compiler to infer how the table is actually used and potentially make better placement decisions for tagets with constraint resources. Entries declared in the P4 source are installed in the table at program load time.

```
tableProperty
   : const ENTRIES '=' '{' entriesList '}' /* immutable entries */

entriesList
 : entry
 | entryList entry

entry
 : keysetExpression ':' actionRef optAnnotations ';'
```

In the current specification, we define entries as immutable (const) – they can only be read, not changed or removed through control plane APIs. It follows that tables that define entries in the P4 source are immutable, thus no additional entries may be inserted in tables that define a set of immutable entries. This design choice is important for the P4 runtime since it does not have to keep track of different types of entries in one table (mutable and immutable). Future versions of the specification may add mutable entries and shall preserve compatibility with the current specification by declaring additional `entries` properties without the `const` keyword.

`keysetExpression` is a tuple that must provide a field for each key in the table keys (see Sec. 12.2.1). The table key type must match the type of the element of the set. `actionRef` must be an action which appears in the table actions list, with all its arguments bound.

Entries in a table are matched in priority order, with the priority computed based on the program order of the entries – entries occuring first have higher priority.

Depending on the `match_kind` of the keys, key set expressions may define one or multiple entries. The compiler will synthesize the correct number of entries to be installed in the table. Target constraints may further restrict the ability of synthesizing entries. For example, if the number of synthesized entries exceeds the table size, the compiler implementation may choose to issue a warning or an error, depending on target capabilities.

Consider the following example:

```
header hdr {
    bit<8>  e;
    bit<16> t;
    bit<8>  l;
    bit<8> r;
    bit<1>  v;
}

struct Header_t {
```

```
    hdr h;
}
struct Meta_t {}

control ingress(inout Header_t h, inout Meta_t m, inout standard_metadata_t standard_meta) {

    action a() { standard_meta.egress_spec = 0; }
    action a_with_control_params(bit<9> x) { standard_meta.egress_spec = x; }

    table t_exact_ternary {

      key = {
            h.h.e : exact;
            h.h.t : ternary;
        }

    actions = {
            a;
            a_with_control_params;
        }

    default_action = a;

    const entries = {
            (0x01, 0x1111 &&& 0xF   ) : a_with_control_params(1);
            (0x02, 0x1181           ) : a_with_control_params(2);
            (0x03, 0x1111 &&& 0xF000) : a_with_control_params(3);
            (0x04, 0x1211 &&& 0x02F0) : a_with_control_params(4);
            (0x04, 0x1311 &&& 0x02F0) : a_with_control_params(5);
            (0x06, _                ) : a_with_control_params(6);
        }
    }

}
```

In this example we define a set of 6 entries that populate various fields in the header and send the invoke one action `a_with_control_params` to send the packet to a particular output port. Once the program is loaded, these entries are installed in the table in the order they are enumerated in the program, with decreasing priority levels.

12.2.1.5. Additional table properties A `table` declaration indicates the interfaces expected from a `table`: keys and actions. However, the best way to implement a table is actually dependent on the nature of the entries that will populate the table at runtime (for example, tables could be dense or sparse, could be implemented as hash-tables, associative memories, tries, etc.) Some architectures may provide additional table properties whose semantics lies outside the scope of this specification. For example, in architectures where table resources are statically allocated, programmers may be required to define a `size` table property, which can be used by the compiler back-end to allocate storage resources. However, an architecture-specific property may not change the semantics of table lookups, which always produce either a `hit` and an action or a `miss`—they can only change how those results are interpreted on the state of the data plane. This restriction is needed to ensure that it is possible to reason about the behavior of tables during compilation.

As another example, an `implementation` property could be used to pass additional information to the compiler back-end. The value of this property could be an instance of an `extern` block

chosen from a suitable library of components. For example, the core functionality of the $P4_{14}$ table `action_profile` constructs could be implemented on architectures that support this feature using a construct such as the following:

```
extern ActionProfile {
   ActionProfile(bit<32> size); // how many distinct actions are expected
}
table t {
    key = { ...}
    size = 1024;
    implementation = ActionProfile(32);  // constructor invocation
}
```

(An action profile specifies the fact that, although the table is expected to have a large number of entries, only a small number of distinct entry values are expected. This can lead to an optimized implementation of the table, using an indirection level to share identical entries. The `ActionProfile` `extern` object in the previous example is meant to convey this information.)

12.2.2. Invoking a table (match-action unit)

A `table` is invoked by calling its `apply` method. Calling an apply method on a table instance returns a value with a `struct` type with two fields. This structure is synthesized by the compiler automatically. For each `table T`, the compiler synthesizes an `enum` and a `struct`, shown in pseudo-P4:

```
enum action_list(T) {
   // one field for each action in the actions list of table T
}
struct apply_result(T) {
    bool hit;
    action_list(T) action_run;
}
```

The evaluation of the `apply` method sets the `hit` field to `true` if a match is found in the lookup-table. This bit can be used to drive the execution of the control-flow in the control block that invoked the table:

```
if (ipv4_match.apply().hit) {
    // there was a hit
} else {
    // there was a miss
}
```

The `action_run` field indicates which kind of action was executed (irrespective of whether it was a hit or a miss). It can be used in a switch statement:

```
switch (dmac.apply().action_run) {
    Drop_action: { return; }
}
```

12.2.3. Match-action unit execution semantics

The semantics of a table invocation statement:

```
m.apply();
```

is given by the following pseudo-code (see also Figure 11):

```
apply_result(m) m.apply() {
    apply_result(m) result;

    var lookupKey = m.buildKey(m.key); // using key block
    action RA = m.table.lookup(lookupKey);
    if (RA == null) {      // miss in lookup table
       result.hit = false;
       RA = m.default_action;  // use default action
    }
    else {
       result.hit = true;
    }
    result.action_run = action_type(RA);
    evaluate_and_copy_in_RA_args(RA);
    execute(RA);
    copy_out_RA_args(RA);
    return result;
}
```

12.3. The Match-Action Pipeline Abstract Machine

We can describe the computational model of a match-action pipeline, embodied by a control block: the body of the control block is executed, similarly to the execution of a traditional imperative program:

- At run-time, statements within a block are executed in the order they appear in the control block.
- Execution of the `return` statement causes immediate termination of the execution of the current `control` block, and a return to the caller.
- Execution of the `exit` statement causes the immediate termination of the execution of the current `control` block and of all the enclosing caller `control` blocks.
- Applying a `table` causes the execution of the corresponding match-action unit, as described above.

12.4. Invoking controls

P4 allows controls to invoke the services of other controls, similar to subroutines. To invoke the services of another control, it must be first instantiated; the services of an instance are invoked by calling it using its `apply` method.

The following example shows a control invocation:

```
control Callee(inout IPv4 ipv4) { ...}
control Caller(inout Headers h) {
     Callee() instance;  // instance of callee
     apply {
          instance.apply(h.ipv4);  // invoke control
     }
}
```

13. Parameterization

In order to support libraries of useful P4 components, both `parser`s and `control` blocks can be additionally parameterized through the use of constructor parameters.

Consider again the parser declaration syntax:

```
parserDeclaration
    : parserTypeDeclaration optConstructorParameters
      '{' parserLocalElements parserStates '}'
    ;

optConstructorParameters
    : /* empty */
    | '(' parameterList ')'
    ;
```

From this grammar fragment we infer that a `parser` declaration can have two sets of parameters:

- The runtime parser parameters
- Optional parser constructor parameters (`optConstructorParameters`)

All constructor parameters must be directionless (i.e., they cannot be `in`, `out` or `inout`). When instantiating a parser one has to supply expressions that can be fully evaluated at compilation time for all `optConstructorParameters`.

Consider the following example:

```
parser GenericParser(packet_in b,           // parser API
                     out Packet_header p)
                     (bool udpSupport) {    // constructor parameters
    EthernetParser() ethParser;
    state start {
        ethParser.apply(b, p.ethernet);
        transition select(p.ethernet.etherType) {
            16w0x0800: ipv4;
        }
    }
    state ipv4 {
        b.extract(p.ipv4);
        transition select(p.ipv4.protocol) {
           6: tcp;
           17: tryudp;
        }
    }
    state tryudp {
        transition select(udpSupport) {
            false: accept;
            true : udp;
        }
    }
    state udp {
         ...
    }
}
```

When instantiating the `GenericParser` one must supply a value for the `udpSupport` parameter, as

in the following example:

```
// TopParser is a GenericParser where udpSupport = false
GenericParser(false) TopParser;
```

13.1. Optional constructor parentheses

For the sake of readability, type definitions with no constructor arguments may omit the second set of parentheses. That is, `control c(...)();` and `control c(...);` are equivalent.

13.2. Direct type invocation

Controls and parsers are often instantiated exactly once. As a light syntactic sugar, control and parser declarations with no constructor parameters may be applied directly, as if they were an instance. This has the effect of creating and applying a local instance of that type.

```
control callee( ... ) { ... }

control caller( ... )( ... ) {
    apply {
        callee.apply( ... ); // callee is treated as an instance
    }
}
```

The definition of `caller` is equivalent to the following.

```
control caller( ... )( ... ) {
    @name("callee") callee() callee_inst; // local instance of callee
    apply {
        callee_inst.apply( ... );         // callee_inst is applied
    }
}
```

This feature is intended to ease the case where a type is instantiated exactly once. For completeness, the behavior of directly invoking the same type more than once is defined as follows.

- Direct type invocation in different scopes will result in different local instances with different fully-qualified control names.
- In the same scope, direct type invocation will result in a different local instance per invocation—however, instances of the same type will share the same global name, via the `@name` annotation. If the type contains controllable entities, then invoking it directly more than once in the same scope is illegal, because it will produce multiple controllable entities with the same fully-qualified control name.

See Section 16.3.2 for details of `@name` annotations.

14. Packet construction (deparsing)

The inverse of parsing is deparsing, or packet construction. P4 does not provide a separate language for packet deparsing; deparsing is done in a `control` block that has at least one parameter of type `packet_out`.

For example, the following code sequence writes first an Ethernet header and then an IPv4 header into a `packet_out`:

```
control TopDeparser(inout Parsed_packet p, packet_out b) {
    apply {
        b.emit(p.ethernet);
```

```
        b.emit(p.ip);
    }
}
```

Emitting a header appends the header to the `packet_out` only if the header is valid. Emitting a header stack will emit all elements of the stack in order of increasing indexes.

14.1. Data insertion into packets

The `packet_out` datatype is defined in the P4 core library, and reproduced below. It provides two methods for appending data to an output packet; both methods are called `emit`. The first version only accepts headers, and the second one accepts arbitrary data.

```
extern packet_out {
    void emit<T>(in T hdr);
    void emit<T>(in bool condition, in T data);
}
```

The first version of `emit` just calls the second one with the header validity bit as a condition.

We describe the meaning of these methods in pseudo-code as follows:

```
packet_out {
    byte[] data;
    unsigned lengthInBits;
    void initializeForWriting() {
        this.data.clear();
        this.lengthInBits = 0;
    }
    // append entire header if it is valid
    // T must be a header type
    void emit<T>(T header) {
        this.emit(header.valid$, header);
    }

    // append the data to the packet if the condition is true
    void emit<T>(bool cond, T data) {
        if (!cond) return;
        this.data.append(data);
        this.lengthInBits += data.lengthInBits;
    }
}
```

We describe the two-argument `emit` method. For a base type `T`, `emit`:

- does nothing if the condition is `false`,
- otherwise it appends the `data` value to the tail of the `packet_out`.

For derived type, `emit` recursively proceeds on fields:

- If the argument is a header, its validity bit is AND-ed with the condition bit to determine; if the result is `false` no action is taken
- If the argument is a header stack, the `emit` statement is applied to each component of the stack starting from the element with index 0.
- If the argument is a `struct` containing multiple fields, the `emit` is recursively applied to each component of the struct in the order of their declaration in the struct.

Appending a bit-string or integer value to a packet_out writes the value starting with the most-significant bit. This process is the inverse of data extraction.

15. Architecture description

The architecture description must be provided by the target manufacturer in the form of a library P4 source file that contains at least one declaration for a package; this package must be instantiated by the user to construct a program for a target. For an example see the Very Simple Switch declaration from Section 5.1.

The architecture description file may pre-define data types, constants, helper package implementations, and errors. It must also declare the types of all the programmable blocks that will appear in the final target: parsers and control blocks. The programmable blocks may optionally be grouped together in packages, which can be nested.

Since some of the target components may manipulate user-defined types, which are unknown at the target declaration time, these are described using type parameters (type variables). This mechanism is similar to the use of generic types or templates in languages such as C++ and Java.

15.1. Example architecture description

The following example describes a switch by using two packages, each containing a parser, a match-action pipeline and a deparser:

```
parser Parser<IH>(packet_in b, out IH parsedHeaders);
// ingress match-action pipeline
control IPipe<T, IH, OH>(in IH inputHeaders,
                         in InControl inCtrl,
                         out OH outputHeaders,
                         out T toEgress,
                         out OutControl outCtrl);
// egress match-action pipeline
control EPipe<T, IH, OH>(in IH inputHeaders,
                         in InControl inCtrl,
                         in T fromIngress,
                         out OH outputHeaders,
                         out OutControl outCtrl);

control Deparser<OH>(in OH outputHeaders, packet_out b);
package Ingress<T, IH, OH>(Parser<IH> p,
                           IPipe<_, IH, OH> map,
                           Deparser<OH> d);
package Egress<T, IH, OH>(Parser<IH> p, Port
                          EPipe<_, IH, OH> map,
                          Deparser<OH> d);
package Switch<T>( // Top-level switch contains two packages
    // type types Ingress.IH and Egress.IH may be different
    Ingress<T, _, _> ingress,
    Egress<T, _, _> egress
);
```

Just from these declarations, even without reading a precise description of the target, the programmer can infer some useful information about the architecture of the described switch, as shown in Figure 12:

- The switch contains two separate packages Ingress and Egress

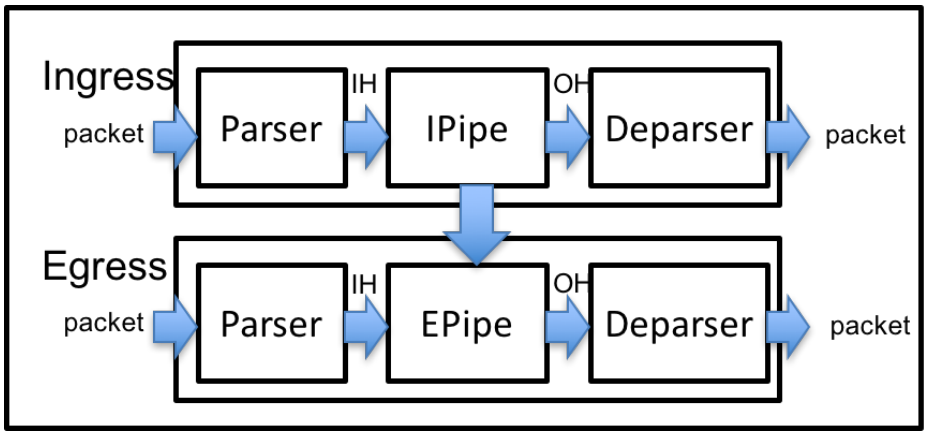

Figure 12. Switch architecture fragment implied by the previous set of declarations.

- The `Parser`, `IPipe` and `Deparser` in the `Ingress` package are chained in this order. The `Ingress.IPipe` block has an input of type `Ingress.IH`, which is an output of the `Ingress.Parser`.
- Similarly, the `Parser`, `EPipe` and `Deparser` are chained in the `Egress` package.
- The `Ingress.IPipe` is connected to the `Egress.EPipe`, because the first one outputs a value of type `T`, which is an input to the second.
- Note that the `Ingress` type `IH` and the `Egress` type `IH` are independent. If we had wanted to show that they are the same type, we would have instead declared `IH` and `OH` at the switch level: `package Switch<IH, OH, T>`.

Note that this architecture models a target switch that contains two separate channels between the ingress and egress pipeline:

- A channel that can pass data directly (through the `T`-type argument – on a software target with shared memory between ingress and egress this could be implemented by passing directly a pointer; on an architecture without shared memory presumably the compiler will need to synthesize automatically serialization code).
- Indirectly through a parser/deparser that serialize data to a packet and back.

15.2. Target program instantiation

In order to build a program for the target, the compiled P4 program has to instantiate a top-level `package` by passing values for all its arguments creating a variable called `main` in the top-level namespace. The types of the arguments have to match the types of the parameters — after a suitable substitution of the type variables. The type substitution can be expressed directly, using type specialization, or can be inferred by a compiler, using a unification algorithm like Hindley-Milner.

For example, given the following type declarations:

```
parser Prs<T>(packet_in b, out T result);
control Pipe<T>(in T data);
package Switch<T>(Prs<T> p, Pipe<T> map);
```

and the following declarations:

```
parser P(packet_in b, out bit<32> index) { ... }
control Pipe1(in bit<32> data) { ... }
control Pipe2(in bit<8> data) { ... }
```

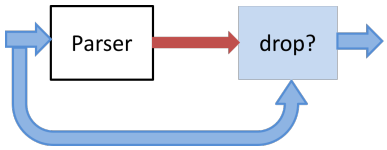

Figure 13. A packet filter target model. The parser computes a Boolean value, which is used to decide whether the packet is dropped.

The following is a legal declaration for the top-level target:

```
Switch(P(), Pipe1()) main;
```

And the following is illegal:

```
Switch(P(), Pipe2()) main;
```

The latter declaration is incorrect because the parser `P` requires `T` to be `bit<32>`, while `Pipe2` requires `T` to be `bit<8>`.

The user can also explicitly specify values for the type variables (otherwise the compiler has to infer values for these type variables):

```
Switch<bit<32>>(P(), Pipe1()) main;
```

15.3. A Packet Filter Model

To illustrate the versatility of P4 architecture description language, we give an example of another architecture, which models a packet filter that makes a drop/no drop decision based only on the computation in a P4 parser, as shown in Figure 13.

This model could be used to program packet filters running in the Linux kernel. For example, we could replace the TCP dump language with the much more powerful P4 language; P4 can support seamlessly new protocols, while providing complete "type safety" during packet processing. For such a target the P4 compiler could generate an eBPF (Extended Berkeley Packet Filter) program, which is injected by the TCP dump utility into the Linux kernel, and executed by the EBPF kernel JIT compiler/runtime.

In this case the target is the Linux kernel, and the architecture model is a packet filter.

The declaration for this architecture is very simple:

```
parser _parser<H>(packet_in packet, out H headers);
control _filter<H>(inout H headers, out bool accept);

package Filter<H>(_parser<H> p, _filter<H> f);
```

16. P4 abstract machine: Evaluation

The evaluation of a P4 program is done in two stages:

- **static evaluation**: at compilation time the P4 program is analyzed and all stateful blocks are instantiated.
- **dynamic evaluation**: at runtime each P4 functional block is executed atomically, in isolation, when it receives control from the architecture

16.1. Compile-time known values

The following are compile-time known values:

- Integer literals, Boolean literals, and string literals.
- Identifiers declared in an `error`, `enum` or `match_kind` declaration.
- The `default` identifier.
- The `size` field of a value with type header stack.
- The `_` identifier when used as a `select` expression label
- Identifiers that represent declared types, actions, tables, parsers, controls or packages.
- List expression where all components are compile-time known values.
- Instances constructed by instance declarations (Section 9.3) and constructor invocations.
- The following expressions (`+, -, *, / , %, cast, !, &, |, &&, ||, << , >> , ~ , >, <, ==, !=, <=, >=, ++, [:]`) when their operands are all compile-time known values.
- Identifiers declared as constants using the `const` keyword.

16.2. Compile-Time Evaluation

Evaluation of a program proceeds in order of declarations, starting in the top-level namespace:

- All declarations (e.g., `parsers`, `controls`, types, constants) evaluate to themselves.
- Each `table` evaluates to a table instance.
- Constructor invocations evaluate to stateful objects of the corresponding type. For this purpose, all constructor arguments are evaluated recursively and bound to the constructor parameters. Constructor arguments must be compile-time known values. The order of evaluation of the constructor arguments should be unimportant — all evaluation orders should produce the same results.
- Instantiations evaluate to named stateful objects.
- The instantiation of a `parser` or `control` block recursively evaluates all stateful instantiations declared in the block.
- The result of the program's evaluation is the value of the top-level `main` variable.

Note that all stateful values are instantiated at compilation time.

As an example, consider the following program fragment:

```
// architecture declaration
parser p(...);
control c(...);
control d(...);

package Switch(p prs, c ctrl, d dep);

extern Checksum16 { ...}

// user code
Checksum16() ck16; // checksum unit instance

parser TopParser(...)(Checksum16 unit) { ...}
control Pipe(...) { ...}
control TopDeparser(...)(Checksum16 unit) { ...}

Switch(TopParser(ck16),
       Pipe(),
       TopDeparser(ck16)) main;
```

The evaluation of this program proceeds as follows:

1. The declarations of `p, c, d, Switch` and `Checksum16` all evaluate as themselves.

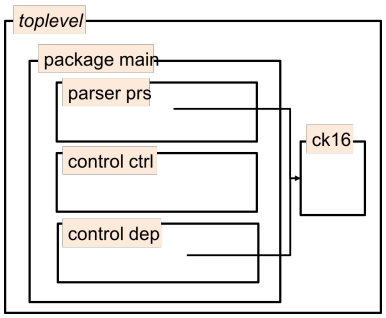

Figure 14. Evaluation result.

2. The `Checksum16() ck16` instantiation is evaluated and it produces an object named `ck16` with type `Checksum16`.
3. The declarations for `TopParser`, `Pipe` and `TopDeparser` evaluate as themselves.
4. The `main` variable instantiation is evaluated:
   (a) The arguments to the constructor are evaluated recursively
   (b) `TopParser(ck16)` is a constructor invocation
   (c) Its argument is evaluated recursively; it evaluates to the `ck16` object
   (d) The constructor itself is evaluated, leading to the instantiation of an object of type `TopParser`
   (e) Similarly, `Pipe()` and `TopDeparser(ck16)` are evaluated as constructor calls.
   (f) All the arguments of the `Switch` package constructor have been evaluated (they are an instance of `TopParser`, an instance of `Pipe`, and an instance of `TopDeparser`). Their signatures are matched with the `Switch` declaration.
   (g) Finally, the `Switch` constructor can be evaluated. The result is an instance of the `Switch` package (that contains a `TopParser` named `prs` — the first parameter of the `Switch`, a `Pipe` named `ctrl` and a `TopDeparser` named `dep`).
5. The result of the program evaluation is the value of the `main` variable, which is the above instance of the `Switch package`.

Figure 14 shows the result of the evaluation in a graphical form. The result is always a graph of instances. There is only one instance of `Checksum16`, called `ck16`, shared between the `TopParser` and `TopDeparser`. Whether this is possible is architecture-dependent. Specific target compilers may require distinct checksum units to be used in distinct blocks.

16.3. Control plane names

Every controllable entity exposed in a P4 program must be assigned a unique, fully-qualified name, which the control plane may use to interact with that entity. The following entities are controllable.

- tables
- keys
- actions
- extern instances

A fully qualified name consists of the local name of a controllable entity prepended with the fully qualified name of its enclosing namespace. Hence, the following program constructs, which enclose controllable entites, must themselves have unique, fully-qualified names.

- control instances
- parser instances

The evaluation process may create multiple instances from one type, each of which must have a unique, fully-qualified name.

16.3.1. Computing control names

The fully-qualified name of a construct is derived by concatenating the fully-qualified name of its enclosing construct with its local name. Constructs with no enclosing namespace, i.e. those defined at the global scope, have the same local and fully-qualified names. The local names of controllable entities and enclosing constructs are derived from the syntax of a P4 program as follows.

16.3.1.1. Tables For each `table` construct, its syntactic name becomes the local name of the table. For example:

```
control c(...)() {
    table t { ... }
}
```

This table's local name is `t`.

16.3.1.2. Keys Syntactically, table keys are expressions. For simple expressions, the local key name can be generated from the expression itself. In the following example, the table `t` has keys with names `data.f1` and `hdrs[3].f2`.

```
table t {
    keys = {
        data.f1 : exact;
        hdrs[3].f2 : exact;
    }
    actions = { ... }
}
```

The following kinds of expressions have local names derived from their syntactic names:

Kind	Example	Name
The `isValid()` method.	`h.isValid()`	`"h.isValid()"`
Array accesses.	`header_stack[1]`	`"header_stack[1]"`
Constants.	`1`	`"1"`
Field projections.	`data.f1`	`"data.f1"`
Slices.	`f1[3:0]`	`"f1[3:0]"`

All other kinds of expressions **must** be annotated with a `@name` annotation (Section 17.1.2), as in the following example.

```
table t {
    keys = {
        data.f1 + 1 : exact @name("f1_inc");
    }
    actions = { ... }
}
```

Here, the `@name("f1_mask")` annotation assigns the local name `"f1_mask"` to this key.

16.3.1.3. Actions For each `action` construct, its syntactic name is the local name of the action. For example:

```
control c(...)() {
    action a(...) { ... }
}
```

This action's local name is `a`.

16.3.1.4. Instances The local names of `extern`, `parser`, and `control` instances are derived based on how the instance is used. If the instance is bound to a name, that name becomes its local control plane name. In the following example, the local name of the instance is `c_inst`.

```
control c(...)() { ... }
c() c_inst;
```

Otherwise, if the instance is created as an actual argument, then its local name is the name of the formal parameter to which it will be bound. In the following example, the local name of the extern instance is `e_in`.

```
extern e( ... ) { ... }
control c ( ... )(e e_in) { ... }
c(e()) c_inst;
```

If the construct being instantiated is passed as an argument to a package, the instance name is derived from the user-supplied type definition when possible. In the following example, the local name of the instance of `MyC` is `c`, and the local name of the extern is `e2`, not `e1`.

```
extern ExternX { ... }
control ArchC(ExternX e1);
package Arch(ArchC c);

control MyC(ExternX e2)() { ... }
Arch(MyC()) main;
```

Note that in this example, the architecture will supply an instance of the extern when it applies the instance of `MyC` passed to the `Arch` package. The fully-qualified name of that instance is `main.c.e2`.

A programmer can always assign a control plane name to an instance by binding it to a name in the program, rather than relying on the parameter name supplied by the type definition or architecture.

```
MyC() d;
Arch(d) main;
```

In the example above, the name of the `MyC` instance is `d`. Next, consider a larger example that demonstrates name generation when there are multiple instances.

```
control Callee() {
    table t { ... }
    apply { t.apply(); }
}
control Caller() {
    Callee() c1;
    Callee() c2;
    apply {
       c1.apply();
       c2.apply();
    }
}
control Simple();
package Top(simple s);
Top(Caller()) main;
```

The compile-time evaluation of this program produces the structure in Figure 15. Notice that there are two instances of the `table t`. These instances must both be exposed to the control plane. To

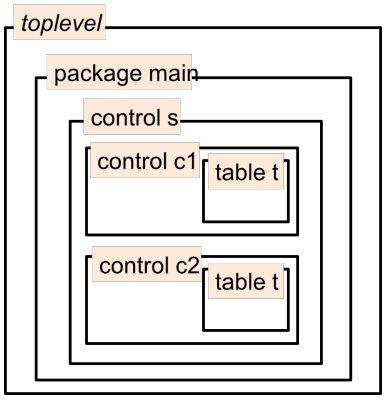

Figure 15. Evaluating a program that has several instantiations of the same component.

name an object in this hierarchy, one uses a path composed of the names of containing instances. In this case, the two tables have names `s.c1.t` and `s.c2.t`, where `s` is the name of the argument to the package instantiation, which is derived from the name of its corresponding formal parameter.

16.3.2. Annotations controlling naming

Control plane-related annotations (Section 17.1.2) can alter the names exposed to the control plane in the following ways.

- The `@hidden` annotation hides a controllable entity from the control plane. This is the only case in which a controllable entity is not required to have a unique, fully-qualified name.
- The `@name` annotation may be used to change the local name of a controllable entity.
- The `@globalname` annotation my be used to change the global name of a controllable entity.

Programs that yield the same fully-qualified name for two different controllable entities are invalid. Care must be taken with `@globalname` in particular: If a type contains a `@globalname` annotation and is instantiated twice, the two instances will have the same fully-qualified name.

16.3.3. Recommendations

The control plane may refer to a controllable entity by a postfix of its fully qualified name when it is unambiguous in the context in which it is used. Consider the following example.

```
control c( ... )() {
    action a ( ... ) { ... }
    table t {
        keys = { ... }
        actions = { a; } }
}
c() c_inst;
```

Control plane software may refer to action `c_inst.a` as `a` when inserting rules into table `c_inst.t`, because it is clear from the definition of the table which action `a` refers to.

Not all unambiguous postfix shortcuts are recommended. Consider the first example in Section 16.3. One might be tempted to refer to `s.c1` simply as `c1`, as no other instance named `c1` appears in the program. However, this leads to a brittle program: Future modifications can never introduce an instance named `c1` or include libraries of P4 code that contain instances with that name.

16.4. Dynamic evaluation

The dynamic evaluation of a P4 program is orchestrated by the target model. Each target model needs to specify the order and the conditions under which the various P4 component programs are dynamically executed. For example, in the Simple Switch example from Section 5.1 the execution flow goes `Parser->Pipe->Deparser`.

Once a P4 execution block is invoked its execution proceeds until termination according to the semantics defined in this document (the various abstract machines).

16.4.1. Concurrency model

A typical packet processing system needs to execute multiple simultaneous logical "threads:" at the very least there is a thread executing the control plane, which can modify the contents of the tables. Architecture specifications should describe in detail the interactions between the control-plane and the data-plane. The data plane can exchange information with the control plane through `extern` function and method calls. Moreover, high throughput packet processing systems may be processing multiple packets simultaneously, e.g., in a pipelined fashion, or concurrently parsing a first packet while performing match-action operations on a second packet. This section specifies the semantics of P4 programs with respect to such concurrent executions.

Each top-level `parser` or `control` block is executed as a separate thread when invoked by the architecture. All the parameters of the block and all local variables are thread-local: i.e., each thread has a private copy of these resources. This applies to the `packet_in` and `packet_out` parameters of parsers and deparsers.

As long as a P4 block uses only thread-local storage (e.g., metadata, packet headers, local variables), its behavior in the presence of concurrency is identical with the behavior in isolation, since any interleaving of statements from different threads must produce the same output.

In contrast, `extern` blocks instantiated by a P4 program are global, shared across all threads. If `extern` blocks mediate access to state (e.g., counters, registers) — i.e., the methods of the `extern` block read and write state, these stateful operations are subject to data races. P4 mandates the following behaviors:

- Execution of an action is atomic, i.e., the other threads can "see" the state as it is either before the start of the action or after the completion of the action.
- Execution of a method call on an extern instance is atomic.

To allow users to express atomic execution of larger code blocks, P4 provides an `@atomic` annotation, which can be applied to block statements, parser states, control blocks or whole parsers.

Consider the following example:

```
extern Register { ... }
control Ingress() {
  Register() r;
  table flowlet { /* read state of r in an action */ }
  table new_flowlet { /* write state of r in an action */ }
  apply {
    @atomic {
       flowlet.apply();
       if (ingress_metadata.flow_ipg > FLOWLET_INACTIVE_TIMEOUT)
          new_flowlet.apply();
}}}
```

This program accesses an extern object `r` of type `Register` in actions invoked from tables `flowlet` (reading) and `new_flowlet` (writing). Without the `@atomic` annotation these two operations would not execute atomically: a second packet may read the state of `r` before the first packet had a chance to update it.

A compiler backend must reject a program containing `@atomic` blocks if it cannot implement the atomic execution of the instruction sequence. In such cases, the compiler should provide reasonable diagnostics.

17. Annotations

Annotations are similar to C# attributes and Java annotations. They are a simple mechanism for extending the P4 language to some limited degree without changing the grammar. To some degree they subsume the C `#pragmas`. Annotations can be added to types, fields, variables, etc. using the `@` syntax (as shown explicitly in the P4 grammar):

```
optAnnotations
    : /* empty */
    | annotations
    ;

annotations
    : annotation
    | annotations annotation
    ;

annotation
    : '@' name
    | '@' name '(' expressionList ')'
    ;
```

17.1. Predefined annotations

Annotation names that start with lowercase letters are reserved for the standard library and architecture. This document pre-defines a set of "standard" annotations. We expect that this list will grow. We encourage custom architectures to define annotations starting with a manufacturer prefix: e.g., company X would use annotations named like `@X_annotation`

17.1.1. Annotations on the table action list

The following two annotations can be used to give additional information to the compiler and control-plane about actions in a table. They have no arguments.

- `@tableonly`: actions with this annotation can only appear within the table, and never as default action.
- `@defaultonly`: actions with this annotation can only appear in the default action, and never in the table.

```
table t {
  actions = {
     a,              // can appear anywhere
     @tableonly b,   // can only appear in the table
     @defaultonly c, // can only appear in the default action
  }
  ...
}
```

17.1.2. Control-plane API annotations

The `@name` annotation directs the compiler to use a different local name when generating the external APIs used to manipulate a language element from the control plane. It must have a string literal argument, which shall not contain the "." character. In the following example, the fully-qualified name of the table is `c_inst.t1`.

```
control c( ... )() {
    @name("t1") table t { ... }
    apply { ... }
}
c() c_inst;
```

The `@globalname` annotation acts like the `@name` annotation, except it overrides the fully-qualified name (not just the local name) for the annotated element, and its argument may contain the "." character. In the following example, the fully-qualified name of the table is `foo.bar`.

```
control c( ... )() {
    @globalname("foo.bar") table t { ... }
    apply { ... }
}
c() c_inst;
```

The `@hidden` annotation hides a controllable entity, eg. a table, key, action, or extern, from the control plane. This effectively removes its fully-qualified name (Section 16.3). It does not take any arguments.

17.1.2.1. Restrictions Each element may be annotated with at most one `@name`, `@globalname`, or `@hidden` annotation, and each control plane name must refer to at most one controllable entity. This is of special concern when using the `@globalname` annotation: If a type containing a `@globalname` annotation is instantiated more than once, it will result in the same global name referring to two controllable entities.

```
control c( ... )() {
    @globalname("foo.bar") table t { ... }
    apply { ... }
}
c() c1;
c() c2;
```

Without the `@globalname` annotation, this program would produce two controllable entities with fully-qualified names `c1.t` and `c2.t`. However, the `@globalname("foo.bar")` annotation renames table `t` in both instances to `foo.bar`, resulting in one name that refers to two controllable entities, which is illegal.

17.1.3. Concurrency control annotations

The `@atomic` annotation, described in Section 16.4.1 can be used to enforce the atomic execution of a code block.

17.2. Target-specific annotations

Each P4 compiler implementation can define additional annotations specific to the target of the compiler. The syntax of the annotations should conform to the above description. The semantics of such annotations is target-specific. They could be used in a similar way to pragmas in the C language.

The P4 compiler should provide:

- Errors when annotations are used incorrectly (e.g., an annotation expecting a parameter but used without arguments, or with arguments of the wrong type
- Warnings for unknown annotations.

A. Appendix: P4 reserved keywords

The following table shows all P4 reserved keywords. Some identifiers are treated as keywords only in specific contexts (e.g., the keyword `actions`).

action	apply	bit	bool
const	control	default	else
enum	error	extern	exit
false	header	if	in
inout	int	match_kind	package
parser	out	return	select
state	struct	switch	table
transition	true	tuple	typedef
varbit	verify	void	

B. Appendix: P4 core library

The P4 core library contains declarations that are useful to most programs.

For example, the core library includes the declarations of the predefined `packet_in` and `packet_out` extern objects, used in parsers and deparsers to access packet data.

```
error {
    NoError,
    PacketTooShort,    // not enough bits in packet for extract
    NoMatch,           // select expression has no matches
    StackOutOfBounds,  // reference to invalid element of a header stack
    HeaderTooShort,    // extracting too many bits into a varbit field
    ParserTimeout      // parser execution time limit exceeded
}
extern packet_in {
    // packet abstraction
    void extract<T>(out T hdr);
    void extract<T>(out T variableSizeHeader,
                    in bit<32> variableFieldSizeInBits);
    T lookahead<T>();
    void advance(in bit<32> sizeInBits);
    bit<32> length(); //packet length in bytes
}
extern packet_out {
    void emit<T>(in T hdr);
    void emit<T>(in bool condition, in T data);
}
action NoAction() {}
match_kind {
    exact,
    ternary,
    lpm
}
```

C. Appendix: Checksums

There are no built-in constructs in $P4_{16}$ for manipulating packet checksums. We expect that all checksum operations can be expressed as `extern` library objects that are provided in target-specific libraries. The standard architecture library should provide such checksum units.

For example, one could provide an incremental checksum unit `Checksum16` (also described in the VSS example in Section 5.2.4) for computing 16-bit one's complement using an `extern` object with a signature such as:

```
extern Checksum16 {
    Checksum16();              // constructor
    void clear();              // prepare unit for computation
    void update<T>(in T data); // add data to checksum
    void remove<T>(in T data); // remove data from existing checksum
    bit<16> get(); // get the checksum for the data added since last clear
}
```

IP checksum verification could be done in a parser as:

```
ck16.clear();           // prepare checksum unit
ck16.update(header.ipv4); // write header
verify(ck16.get() == 16w0, error.IPv4ChecksumError); // check for 0 checksum
```

IP checksum generation could be done as:

```
header.ipv4.hdrChecksum = 16w0;
ck16.clear();
ck16.update(header.ipv4);
header.ipv4.hdrChecksum = ck16.get();
```

Moreover, some switch architectures do not perform checksum verification, but only update checksums incrementally to reflect packet modifications. This could be achieved as well, as the following P4 program fragments illustrates:

```
ck16.clear();
ck16.update(header.ipv4.hdrChecksum);  // original checksum
ck16.remove( { header.ipv4.ttl, header.ipv4.proto } );
header.ipv4.ttl = header.ipv4.ttl - 1;
ck16.update( { header.ipv4.ttl, header.ipv4.proto } );
header.ipv4.hdrChecksum = ck16.get();
```

D. Appendix: Open Issues

There are a number of open issues that are currently under discussion in the P4 design working group. A brief summary of these issues is highlighted in this section. We seek input on these issues from the community, and encourage experimenting with different implementations in the compiler before converging on the specification.

D.1. Portable Switch Architecture

Portability and composability are critical to P4's long-term success: - Composability: implement different features, such as In-band Network Telemetry (INT), Network Virtualization, and Load Balancing in separate P4 programs written for the PSA, should easily interoperate when invoked from a toplevel program. - Portability: a P4 implementation of a certain function, such as INT, against PSA should work consistently across architectures that support the PSA. To that end, a specification for an architecture definition that enables programmers to write composable P4

programs, called the *Portable Switch Architecture*, is being developed by the Architecture Working Group.

D.2. Generalized switch statement behavior

P4$_{16}$ includes both `switch` statements 10.7 and `select` expressions 11.6. There are real differences in the current version of the language – expression vs. statement, and the latter must evaluate to a state value.

We propose generalizing `switch` statements to match the design used in most programming language: a multi-way conditional that executes the first branch that matches from a list of cases.

```
switch(e1,...,en) {
  pat_1 : stmt1;
  ...
  pat_m : stmtm;
}
```

Here, the value being scrutinized is given by a tuple `(e1,...,en)`, and the patterns are given by expressions that denote sets of values. The value matches a branch if it is an element of the set denoted by the pattern. Unlike C and C++, there is no break statement so control "falls through" to the next case only when there is no statement associated with the case label.

This design is intended to capture the standard semantics of `switch` statements as well as a common idiom in P4 parsers where they are used to control transitions to different parser states depending on the values of one or more previously-parsed values. Using `switch` statements, we can also generalize the design for parsers, eliminating select and lifting most restrictions on which kinds of statements may appear in a state. In particular, we allow conditional statements and `select` statements, which may be nested arbitrarily. This language can be translated into more restricted versions, where the body of each state comprised a sequence of variable declarations, assignments, and method invocations followed by a single`transition` statement by introducing new states.

We also generalize the design for processing of table hit/miss and actions in control blocks, by generating implicit types for actions and results.

The counter-argument to this proposal is that the semantics of `select` in the parser is sufficiently distinct from the `switch` statement, and moreover these are constructs that network programmers are already faminliar with, and they are typically mapped very efficiently onto a variety of targets.

D.3. Undefined behaviors

The presence of undefined behavior has caused numerous problems in languages like C and HTML, including bugs and serious security vulnerabilities. There are a few places where evaluating a P4 program can result in undefined behaviors: out parameters, uninitialized variables, accessing header fields of invalid headers, and accessing header stacks with an out of bounds index. We think we should make every attempt to avoid undefined behaviors in P4$_{16}$, and therefore we propose to strengthen the wording in the specification, such that by default, we rule out programs that exhibit the behaviors mentioned above. Given the concern for performance, we propose to define compiler flags and/or pragmas that can override the safe behavior. However, our expectation is that programmers should be guided toward writing safe programs, and encouraged to think harder when excepting from the safe behavior.

D.4. Structured Iteration

Introducing a `foreach` style iterator for operating over header stacks will alleviate the need of using C preprocessor directives to specify the size of header stacks.

For example:

```
foreach hdr in hdrs {
  ... operations over HDR ...
}
```

Since the stacks are always known statically (at compile-time), the compiler could transform the `foreach` statement into the replicated code with explicit index references at compile-time. This has the advantage of allowing the code to be written without regard to a parameterized header stack length.

Since the compiler can statically determine the number of operations that would result from the `foreach` it can also reject a program if the result requires more action resources than are available, or can split the action code up to fit available resources as needed.

E. Appendix: P4 grammar

This is the grammar of $P4_{16}$ written using the YACC/bison language. Absent from this grammar is the precedence of various operations.

The grammar is actually ambiguous, so the lexer and the parser must collaborate for parsing the language. In particular, the lexer must be able to distinguish two kinds of identifiers:

- Type names previously introduced (`TYPE` tokens)
- Regular identifiers (`IDENTIFIER` token)

The parser has to use a symbol table to indicate to the lexer how to parse subsequent appearances of identifiers. For example, given the following program fragment:

```
typedef bit<4> t;
struct s { ...}
t x;
parser p(bit<8> b) { ... }
```

The lexer has to return the following terminal kinds:

```
t - TYPE
s - TYPE
x - IDENTIFIER
p - TYPE
b - IDENTIFIER
```

This grammar has been heavily influenced by limitations of the Bison parser generator tool.

Several other constant terminals appear in these rules:

```
- SHL is <<
- LE is <=
- GE is >=
- NE is !=
- EQ is ==
- PP is ++
- AND is &&
- OR is ||
- MASK is &&&
- RANGE is ..
- DONTCARE is _
```

The `STRING_LITERAL` token corresponds to a string literal enclosed within double quotes, as described in Section 6.3.3.3.

All other terminals are uppercase spellings of the corresponding keywords. For example, `RETURN` is the terminal returned by the lexer when parsing the keyword return.

```
p4program
    : /* empty */
    | p4program declaration
    | p4program ';'  /* empty declaration */
    ;

declaration
    : constantDeclaration
    | externDeclaration
    | actionDeclaration
    | parserDeclaration
    | typeDeclaration
    | controlDeclaration
    | instantiation
    | errorDeclaration
    | matchKindDeclaration
    ;

nonTypeName
    : IDENTIFIER
    | APPLY
    | KEY
    | ACTIONS
    | STATE
    ;

name
    : nonTypeName
    | TYPE
    | ERROR
    ;

optAnnotations
    : /* empty */
    | annotations
    ;

annotations
    : annotation
    | annotations annotation
    ;

annotation
    : '@' name
    | '@' name '(' expressionList ')'
    ;

parameterList
    : /* empty */
    | nonEmptyParameterList
    ;
```

```
nonEmptyParameterList
    : parameter
    | nonEmptyParameterList ',' parameter
    ;

parameter
    : optAnnotations direction typeRef name
    ;

direction
    : IN
    | OUT
    | INOUT
    | /* empty */
    ;

packageTypeDeclaration
    : optAnnotations PACKAGE name optTypeParameters
      '(' parameterList ')'
    ;

instantiation
    : typeRef '(' argumentList ')' name ';'
    : annotations typeRef '(' argumentList ')' name ';'
    ;

optConstructorParameters
    : /* empty */
    | '(' parameterList ')'
    ;

dotPrefix
    : '.'
    ;

/************************** PARSER *****************************/

parserDeclaration
    : parserTypeDeclaration optConstructorParameters
      /* no type parameters allowed in the parserTypeDeclaration */
      '{' parserLocalElements parserStates '}'
    ;

parserLocalElements
    : /* empty */
    | parserLocalElements parserLocalElement
    ;

parserLocalElement
    : constantDeclaration
    | variableDeclaration
```

```
    | instantiation
    ;

parserTypeDeclaration
    : optAnnotations PARSER name optTypeParameters '(' parameterList ')'
    ;

parserStates
    : parserState
    | parserStates parserState
    ;

parserState
    : optAnnotations STATE name '{' parserStatements transitionStatement '}'
    ;

parserStatements
    : /* empty */
    | parserStatements parserStatement
    ;

parserStatement
    : assignmentOrMethodCallStatement
    | directApplication
    | parserBlockStatement
    | constantDeclaration
    | variableDeclaration
    ;

parserBlockStatement
    : optAnnotations '{' parserStatements '}'
    ;

transitionStatement
    : /* empty */
    | TRANSITION stateExpression
    ;

stateExpression
    : name ';'
    | selectExpression
    ;

selectExpression
    : SELECT '(' expressionList ')' '{' selectCaseList '}'
    ;

selectCaseList
    : /* empty */
    | selectCaseList selectCase
    ;
```

```
selectCase
    : keysetExpression ':' name ';'
    ;

keysetExpression
    : tupleKeysetExpression
    | simpleKeysetExpression
    ;

tupleKeysetExpression
    : '(' simpleKeysetExpression ',' simpleExpressionList ')'
    ;

simpleExpressionList
    : simpleKeysetExpression
    | simpleExpressionList ',' simpleKeysetExpression
    ;

simpleKeysetExpression
    : expression
    | DEFAULT
    | DONTCARE
    | expression MASK expression
    | expression RANGE expression
    ;

/************************* CONTROL ***********************/

controlDeclaration
    : controlTypeDeclaration optConstructorParameters
      /* no type parameters allowed in controlTypeDeclaration */
      '{' controlLocalDeclarations APPLY controlBody '}'
    ;

controlTypeDeclaration
    : optAnnotations CONTROL name optTypeParameters
      '(' parameterList ')'
    ;

controlLocalDeclarations
    : /* empty */
    | controlLocalDeclarations controlLocalDeclaration
    ;

controlLocalDeclaration
    : constantDeclaration
    | actionDeclaration
    | tableDeclaration
    | instantiation
    | variableDeclaration
    ;
```

```
controlBody
    : blockStatement
    ;

/************************* EXTERN ************************/

externDeclaration
    : optAnnotations EXTERN nonTypeName optTypeParameters '{' methodPrototypes '}'
    | optAnnotations EXTERN functionPrototype ';'
    ;

methodPrototypes
    : /* empty */
    | methodPrototypes methodPrototype
    ;

functionPrototype
    : typeOrVoid name optTypeParameters '(' parameterList ')'
    ;

methodPrototype
    : functionPrototype ';'
    | TYPE '(' parameterList ')' ';'
    ;

/************************* TYPES *************************/

typeRef
    : baseType
    | typeName
    | specializedType
    | headerStackType
    ;

prefixedType
    : TYPE
    | dotPrefix TYPE
    ;

typeName
    : prefixedType
    ;

tupleType
    : TUPLE '<' typeArgumentList '>'
    ;

headerStackType
    : typeName '[' expression ']'
    ;

specializedType
```

```
    : prefixedTYpe '<' typeArgumentList '>'
    ;

baseType
    : BOOL
    | ERROR
    | BIT
    | BIT '<' INTEGER '>'
    | INT '<' INTEGER '>'
    | VARBIT '<' INTEGER '>'
    ;

typeOrVoid
    : typeRef
    | VOID
    | nonTypeName     // may be a type variable
    ;

optTypeParameters
    : /* empty */
    | '<' typeParameterList '>'
    ;

typeParameterList
    : nonTypeName
    | typeParameterList ',' nonTypeName
    ;

typeArg
    : DONTCARE
    | typeRef
    ;

typeArgumentList
    : typeArg
    | typeArgumentList ',' typeArg
    ;

typeDeclaration
    : derivedTypeDeclaration
    | typedefDeclaration
    | parserTypeDeclaration ';'
    | controlTypeDeclaration ';'
    | packageTypeDeclaration ';'
    ;

derivedTypeDeclaration
    : headerTypeDeclaration
    | headerUnionDeclaration
    | structTypeDeclaration
    | enumDeclaration
    ;
```

```
headerTypeDeclaration
    : optAnnotations HEADER name '{' structFieldList '}'
    ;

headerUnionDeclaration
    : optAnnotations HEADER_UNION name { structure.declareType(*$3); }
      '{' structFieldList '}'          { $$ = new IR::Type_Union(@3, *$3, $1, *$6); }
    ;

structTypeDeclaration
    : optAnnotations STRUCT name '{' structFieldList '}'
    ;

structFieldList
    : /* empty */
    | structFieldList structField
    ;

structField
    : optAnnotations typeRef name ';'
    ;

enumDeclaration
    : optAnnotations ENUM name '{' identifierList '}'
    ;

errorDeclaration
    : ERROR '{' identifierList '}'
    ;

matchKindDeclaration
    : MATCH_KIND '{' identifierList '}'
    ;

identifierList
    : name
    | identifierList ',' name
    ;

typedefDeclaration
    : annotations TYPEDEF typeRef name ';'
    | TYPEDEF typeRef name ';'
    | annotations TYPEDEF derivedTypeDeclaration name ';'
    | TYPEDEF derivedTypeDeclarationname ';'
    ;

/************************* STATEMENTS ************************/

assignmentOrMethodCallStatement
    : lvalue '(' argumentList ')' ';'
    | lvalue '<' typeArgumentList '>' '(' argumentList ')' ';'
```

```
    | lvalue '='  expression ';'
    ;

emptyStatement
    : ';'
    ;

returnStatement
    : RETURN ';'
    ;

exitStatement
    : EXIT ';'
    ;

conditionalStatement
    : IF '(' expression ')' statement
    | IF '(' expression ')' statement ELSE statement
    ;

// To support direct invocation of a control or parser without instantiation
directApplication
    : typeName '.' APPLY '(' argumentList ')' ';'

statement
    : assignmentOrMethodCallStatement
    | directApplication
    | conditionalStatement
    | emptyStatement
    | blockStatement
    | exitStatement
    | returnStatement
    | switchStatement
    ;

blockStatement
    : optAnnotations '{' statOrDeclList '}'
    ;

statOrDeclList
    : /* empty */
    | statOrDeclList statementOrDeclaration
    ;

switchStatement
    : SWITCH '(' expression ')' '{' switchCases '}'
    ;

switchCases
    : /* empty */
    | switchCases switchCase
    ;
```

```
switchCase
    : switchLabel ':' blockStatement
    | switchLabel ':'
    ;

switchLabel
    : name
    | DEFAULT
    ;

statementOrDeclaration
    : variableDeclaration
    | constantDeclaration
    | statement
    | instantiation
    ;

/************ TABLES *************/
tableDeclaration
    : optAnnotations TABLE name '{' tablePropertyList '}'
    ;

tablePropertyList
    : tableProperty
    | tablePropertyList tableProperty
    ;

tableProperty
    : KEY '=' '{' keyElementList '}'
    | ACTIONS '=' '{' actionList '}'
    | CONST ENTRIES '=' '{' entriesList '}' /* immutable entries */
    | optAnnotations CONST IDENTIFIER '=' initializer ';'
    | optAnnotations IDENTIFIER '=' initializer ';'
    ;

keyElementList
    : /* empty */
    | keyElementList keyElement
    ;

keyElement
    : expression ':' name optAnnotations ';'
    ;

actionList
    : actionRef ';'
    | actionList actionRef ';'
    ;

entriesList
    : entry
```

```
    | entryList entry

entry
    : optAnnotations keysetExpression ':' actionRef ';'

actionRef
    : optAnnotations name
    | optAnnotations name '(' argumentList ')'
    ;

/************************ ACTION *******************************/

actionDeclaration
    : optAnnotations ACTION name '(' parameterList ')' blockStatement
    ;

/************************ VARIABLES ****************************/

variableDeclaration
    : annotations typeRef name optInitializer ';'
    | typeRef name optInitializer ';'
    ;

constantDeclaration
    : optAnnotations CONST typeRef name '=' initializer ';'
    ;

optInitializer
    : /* empty */
    | '=' initializer
    ;

initializer
    : expression
    ;

/************************ Expressions **************************/

argumentList
    : /* empty */
    | nonEmptyArgList
    ;

nonEmptyArgList
    : argument
    | nonEmptyArgList ',' argument
    ;

argument
    : expression
    ;
```

```
expressionList
    : expression
    | expressionList ',' expression
    ;

member
    : name
    ;

prefixedNonTypeName
    : nonTypeName
    | dotPrefix nonTypeName
    ;

lvalue
    : prefixedNonTypeName
    | lvalue '.' member
    | lvalue '[' expression ']'
    | lvalue '[' expression ':' expression ']'
    ;

%left ','
%nonassoc '?'
%nonassoc ':'
%left OR
%left AND
%left '|'
%left '^'
%left '&'
%left EQ NE
%left '<' '>' LE GE
%left SHL
%left PP '+' '-'
%left '*' '/' '%'
%right PREFIX
%nonassoc ']' '(' '['
%left '.'

// Additional precedences need to be specified

expression
    : INTEGER
    | TRUE
    | FALSE
    | STRING_LITERAL
    | nonTypeName
    | '.' nonTypeName
    | expression '[' expression ']'
    | expression '[' expression ':' expression ']'
    | '{' expressionList '}'
    | '(' expression ')'
    | '!' expression
```

```
    | '~' expression
    | '-' expression
    | '+' expression
    | typeName '.' member
    | ERROR '.' member
    | expression '.' member
    | expression '*' expression
    | expression '/' expression
    | expression '%' expression
    | expression '+' expression
    | expression '-' expression
    | expression SHL expression        // <<
    | expression '>''>' expression     // check that >> are adjacent
    | expression LE expression         // <=
    | expression GE expression         // >=
    | expression '<' expression
    | expression '>' expression
    | expression NE expression         // !=
    | expression EQ expression         // ==
    | expression '&' expression
    | expression '^' expression
    | expression '|' expression
    | expression PP expression         // ++
    | expression AND expression        // &&
    | expression OR expression         // ||
    | expression '?' expression ':' expression
    | expression '<' typeArgumentList '>' '(' argumentList ')'
    | expression '(' argumentList ')'
    | typeRef '(' argumentList ')'
    | '(' typeRef ')' expression
    ;
```